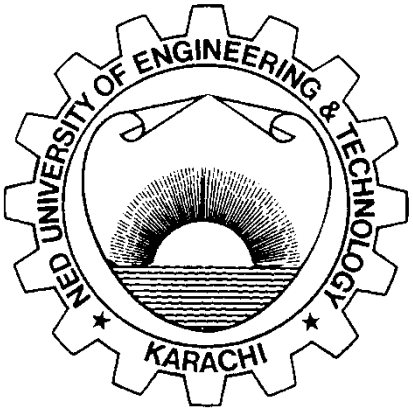
Practical Workbook
Digital Communication Systems



Name : _____

Semester : _____ Batch: _____

Roll No. : _____

Group No. : _____

Department : _____

Dept. of Computer & Information Systems Engineering

NED University of Engineering & Technology,

Karachi – 75270, Pakistan

Practical Workbook
Digital Communication Systems

Prepared By	:	Mr. Saad Qasim (Lecturer)
Supervised By	:	Mr. Syed Jaseemuddin (Lecturer)
Reviewed By	:	Dr. Ing. Shehzad Hasan (Assistant Professor) Mr. Shahab Tahzeeb (Assistant Professor)

First Edition - 2011

Dept. of Computer & Information Systems Engineering
NED University of Engineering & Technology,
Karachi – 75270, Pakistan

CONTENTS

Lab Session No.	Object	Page No.
Section One :	**Basics of Digital Communication Systems**	1
1	Introduction to Digital Communication Systems	2
2	Signals and Spectra-*I*	8
3	Signals and Spectra-*II*	14
4	Pulse Amplitude Modulation	20
5	Familiarization with Pulse Code Modulation	24
Section Two :	**Modulation and Demodulation Techniques in Bandpass**	29
6	Introduction to Modulation techniques	30
7	Understanding Demodulation techniques	38
Section Three :	**Channel Coding**	45
8	Encoding Linear Block Codes	46
9	Decoding Linear Block Codes	51
10	Introduction to Cyclic Codes	64
11	Familiarization with Huffman Coding Techniques	81
Section Four :	**Signal Processing**	88
12	Audio Signal Processing	89
Appendix A :	**An introduction to MATLAB**	93
Appendix B :	**An introduction to Simulink**	105

Section One

Basics of
Digital Communication Systems

Lab Session 01

Introduction to Digital Communication Systems

OBJECTIVE

Getting familiar with the basics of digital communication systems.

THEORY

The mathematical foundation for digital communications was established in the two classic papers by Claude Shannon (1948a, b). In this pioneering work he formulated the basic problem of reliable transmission of information in statistical terms, using probabilistic models for information sources and communication channels. Based on such a statistical formulation, he adopted a logarithmic measure for the information content of a source and established basic limits on the maximum rate that digital information can be transmitted reliably over a communication channel. Shannon's work gave birth to a new field that is now called *information theory.*

Initially the fundamental work of Shannon had little impact on the design and development of new digital communications systems. In part, this was due to the relatively small demand for digital information transmission during the decade of the 1950s. Another reason was the relatively large complexity and high cost of digital hardware required to achieve the high efficiency and the high reliability predicted by Shannon's theory. Shannon's work, however, served as stimulation for many researchers to refine and to extend his results in a number of different ways. As a consequence, the theory that has developed to date is widely applied in the design of digital communication systems.

Nyquist–Shannon sampling theorem

The **Nyquist–Shannon sampling theorem**, which has been named after Harry Nyquist and Claude Shannon, is a fundamental result in the field of information theory, in particular telecommunications and signal processing. Sampling is the process of converting a signal which is a function of continuous time or space into function of discrete time or space. Shannon's version of the theorem states:

> *If a function x(t) contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced 1/(2B) seconds apart.*
>
> *Or*
>
> *The sampling frequency should be at least twice the highest frequency contained in the signal.*
>
> *Mathematically*
>
> $$fs > 2\,fc$$
>
> *where fs is the sampling frequency (how often samples are taken per unit of time or space), and fc is the highest frequency contained in the signal.*

Sampling a sound (using a microphone) or image (using a CCD camera) can introduce a distortion into the signal called "aliasing." Aliasing takes many forms; for example the apparent backward motion of wagon wheels in western movies, which occurs because the movie is actually a sequence of still frames. Nyquist showed that if the sampling rate is twice the highest frequency component, then the original information can be reconstructed without aliasing.

Why Digital Modulation?

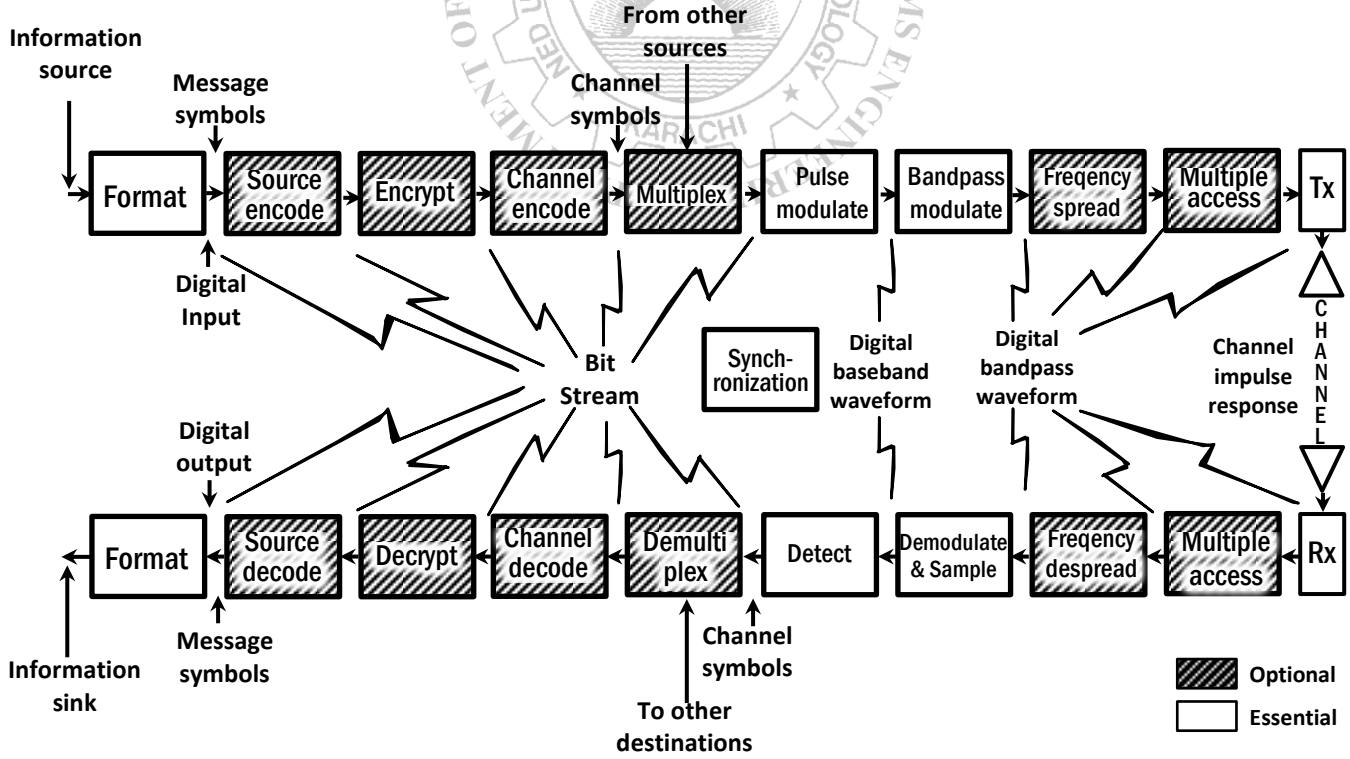
The move to digital modulation provides more information capacity, compatibility with digital data services, higher data security, better quality communications, and quicker system availability. Developers of communications systems face these constraints:

- Available bandwidth
- Permissible power
- Inherent noise level of the system

The RF spectrum must be shared, yet every day there are more users for that spectrum as demand for communications services increases. Digital modulation schemes have greater capacity to convey large amounts of information than analog modulation schemes.

Modules of Digital Communication System:

Digital communication system can be modularized as shown in the following figure.



Digital Baseband waveform:

In telecommunications and signal processing, ***baseband*** is an adjective that describes signals and systems whose range of frequencies is measured from close to 0 hertz to a cut-off frequency, a maximum bandwidth or highest signal frequency; it is sometimes used as a noun for a band of frequencies starting close to zero. Baseband can often be considered as a synonym to **lowpass** or **non-modulated**, and antonyms to passband, bandpass, carrier-modulated or radio frequency (RF) signal.

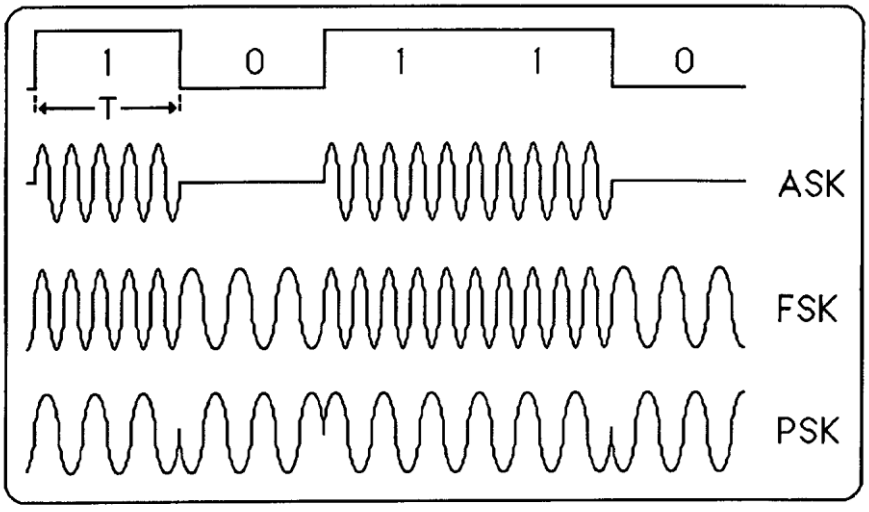
A *baseband bandwidth* is equal to the highest frequency of a signal or system, or an upper bound on such frequencies, for example the upper cut-off frequency of a passband filters. By contrast, a passband bandwidth is the difference between a highest frequency and a nonzero lowest frequency.

A *baseband channel* or *lowpass channel* (or *system*, or *network*) is a communication channel that can transfer frequencies that are close to zero. Examples are serial cables and local area networks (LANs), as opposed to passband channels such as radio frequency channels and passband filtered wires of the analog telephone network. Frequency division multiplexing (FDM) allows an analog telephone wire to carry a baseband telephone call, concurrently as one or several carrier-modulated telephone calls.

Baseband transmission, also known as *Baseband modulation* or line coding, aims at transferring a digital bit stream over a baseband channel, as an alternative to passband transmission, also known as *carrier-modulated* transmission.

Digital Bandpass waveform:

Bandpass modulation is the process by which an information signal is transformed into a sinusoidal waveform. Since a sinusoid has three features; *amplitude*, *phase* and *frequency*, bandpass modulation can be defined as the process by which the amplitude, phase, frequency of an RF carrier, or a combination of them, is varied in accordance with the information signal to be transmitted.



Tool for Analysing Behaviour of Digital Systems & Signals:

MATLAB is used as a primary tool for analyzing different signals and communication techniques in all proceeding lab sessions. A reference to MATLAB is provided in Appendix A and reference to Simulink is also included in the workbook as Appendix B.

EXERCISE:

1.1 What do you know about Nyquist Sampling Theorem

1.2 Why digital modulation is preferred over analog? Explain in detail

1.3 What is the difference between Baseband and Bandpass waveform? Explain

1.4 What are the most fundamental digital modulation techniques based on keying? Explain.

Problems with reference to Appendix A:

1.5 Plot the following three curves:

2 cos(t), sin(t) and cos(t)+sin(t).

Use a time period such that 2 or 3 peaks occur for each curve. Use solid, dashed, and '+' symbols for the different curves. Attach graph at the last page. Write down your observation

1.6 Calculate the rank, determinant and matrix inverse of the following matrices
(use `help rank`, `help det` and `help inv`)

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 1 \\ -1 & -2 & -1 \\ 2 & 4 & 2 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 1 & 2 & 1 \\ -1 & 4 & -1 \\ 2 & 4 & 2 \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} 1 & 2 & 1 \\ -1 & 4 & -1 \\ 2 & 4 & 5 \end{pmatrix}$$

1.7 Find $\mathbf{C}\mathbf{C}^{-1}$; use same value of **C** as given in **Exercise 1.6**

Attach your graph (for Exercise 1.5) here.

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 02

Signals and Spectra I

OBJECTIVE

Getting familiar with the representation of signals in MATLAB

THEORY

MATLAB is very useful for making scientific and engineering plots. You can create plots of known, analytical functions, you can plot data from other sources such as experimental measurements, and you can analyze data, perhaps by fitting it to a curve, and then plot a comparison. MATLAB also has powerful built-in routines for drawing contour and three-dimensional surface plots.

Signal's representation

A signal in MATLAB is represented by a row vector:

- Examples:
 - x = [2, 3, -5, -3, 1]
 - n = 2:17 = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17]

 Default step size is 1
 - n = 2:**3**:17 = [2, 5, 8, 11, 14, 17]

'3' Step size

Plotting in MATLAB

A vector is plotted against a vector

- lengths of vectors must match

Two functions

- **plot**
 - for **CT** signals
- **stem**
 - for **DT** signals

The basic `plot` command

Two-dimensional line and symbol plots are created with the `plot` command. In its simplest form `plot` takes two arguments

```
>> plot(xdata,ydata)
```

Where `xdata` and `ydata` are vectors containing the data. Note that `xdata` and `ydata` must be the same length and both must be the same type, i.e., both must be either row or column vectors. Additional arguments to the plot command provide other options including the ability to plot multiple data sets, and a choice of colors, symbols and line types for the data.

Example

Plot the following signal

$x = 10\sin \pi t$ (**vector x** against **vector t)**

- ✓ must decide on vector's lengths
 - t = [-2:0.002:2]
- ✓ How to generate vector x?
 - x = 10 * sin (pi * t)
 - plot(**t**, **x**)

Vector on X-axis | Vector on Y-axis

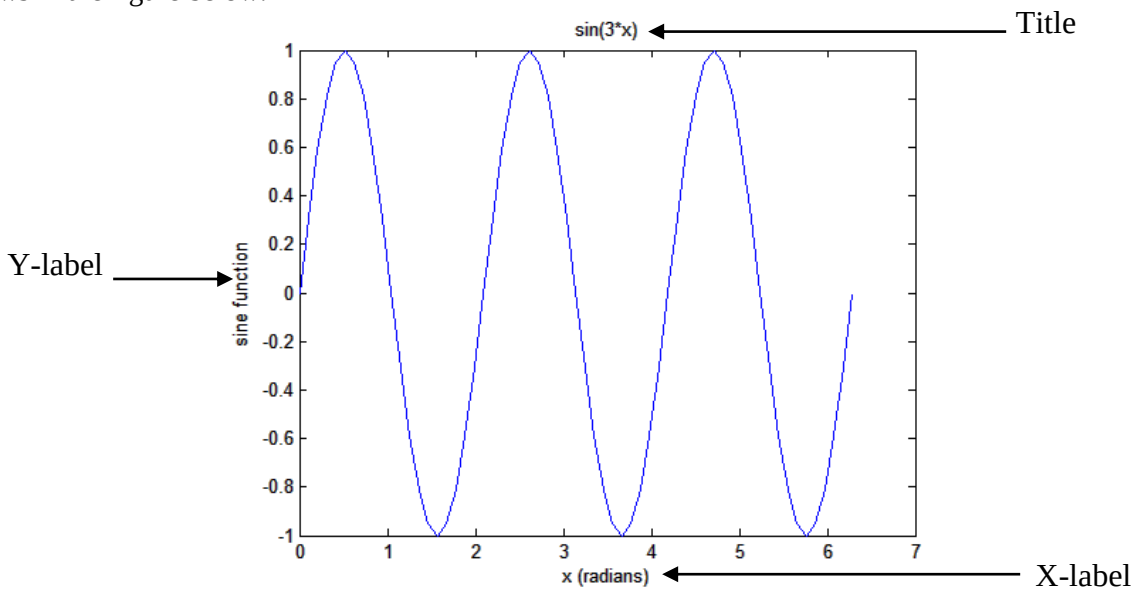
- plot(t, x) ←→ plot([-2:0.002:2], 10 * sin(pi*[-2:0.002:2]))
 - instead of defining "t" & "x" separately, you can directly pass their values/range
- title('Example Sinusoid')
 - to name the plot use "title" command
- xlabel('time(sec)')
- ylabel('Amplitude') } to label x-axis and y-axis

A simple line plot

Here are the MATLAB commands to create a simple plot of y = sin (3*pi*x) from 0 to 2*pi.

```
>> x = 0:pi/30:2*pi;          % x vector, 0 <= x <= 2*pi, increments of
                              pi/30
>> y = sin(3*x);              % vector of y values
>> plot(x,y)                  % create the plot
>> xlabel('x (radians)');     % label the x-axis
>> ylabel('sine function');   % label the y-axis
>> title('sin(3*x)');         % put a title on the plot
```

The effect of the labelling commands, `xlabel`, `ylabel`, and `title` are indicated by the text and arrows in the figure below.



Changing symbol or line types

The symbol or line type for the data can be changed by passing an optional third argument to the plot command. For example

```
>> plot(x,y,'o');
```

plots data in the x and y vectors using circles drawn in the default color (yellow), and

```
>> plot(x,y,'r:');
```

plots data in the x and y vectors by connecting each pair of points with a red dashed line.

The third argument of the plot command is a one, two or three character string of the form 'cs', where 'c' is a single character indicating the color and 's' is a one or two character string indicating the type of symbol or line. The color selection is optional. Allowable color and symbols types are summarized in appendix-A. Refer to "help plot" for further information.

DT Plots

Plot the DT sequences:

- x = [2, 3, -1, 5, 4, 2, 3, 4, 6, 1]
- stem(x);

- x = [2, 3, -1, 5, 4, 2, 3, 4, 6, 1]
- n = -6:3;
- stem(n, x);

Zero & One Vectors

- zeros(1, 5)
 - [0 0 0 0 0]
- ones(1, 5)
 - [1 1 1 1 1]

EXERCISE

2.1 Use MATLAB to draw the graph of $f(x) = x^2 - 2x - 3$ on the interval $[-1, 3]$.

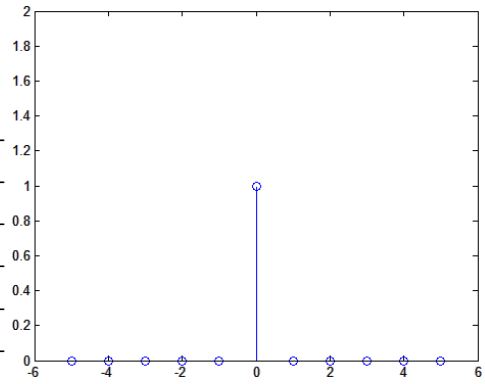
2.2 Generate following elementary DT signals:

– Unit Impulse
– Unit Step
– Unit Ramp
– Exponential Sequence

Example:

– Unit Impulse

```
n = -5:5;
stem (n, [zeros(1,5) 1 zeros(1,5)])
```



– Unit Step

– Unit Ramp

– Exponential sequence

2.3 Generate the following signal for -10<n<10

$$x = (10\cos\pi n)(5\cos 0.25\pi n)$$

Attach your graph (for Exercise 2.1 & 2.3) here.

NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
KARACHI

Lab Session 03

Signals and Spectra II

OBJECTIVE

The objective of this lab session is to get familiarize with different continuous wave signals and to plot time domain graphs for such signals. The main objectives of this lab are:

1. Creating and plotting different continuous time waves.
2. Plotting periodic signals.
3. Integration of functions and plotting them.
4. Impulse function (Dirac Delta) plotting.
5. Performing simple operations on signals.

THEORY

Introduction

- A **signal** is a form in which data is transmitted. It describes the behaviour of data.
- Mathematically represented by a dependent variable (e.g. *x(t)*) and independent variable (in this case *t*).

$$x(t) = t^2 + 1$$

- Special type of signal is the sinusoid wave which is a combination of sine and cosine waves. These signals are easy to visualize, generate and process. Any signal could be represented by the sum of these sinusoidal waves.
- There are three parameters mainly that describe a signal: Amplitude, frequency and phase. Amplitude describes the strength or intensity of the signal at a given time. Frequency means the number of cycle or repetition per second. This definition is only valid for periodic signals since periodic signals repeat themselves every T second (known as the period time duration). For non-periodic signals, these are converted or represented by periodic sinusoid signals. Each sinusoid signal represents a single frequency.
- A typical sinusoid signal is given by: $x(t) = A\sin(2\pi ft + \phi)$
- Where ϕ represents the phase of the signal which describes the position of the signal w.r.t time. A is the amplitude of the signal which describes the intensity of the signal at a given time. f_0 is the frequency which is the number of cycles per second.

Signal classifications:

1. **Periodic and Non-periodic (Aperiodic):**

- Periodic signals are those signals that repeat themselves over a period of time known as the period of the signal (e.g. sine function) and given by:

$$T_0 = 1/f_0 \quad \text{and} \quad x(t) = x(t + T_0)$$

- Aperiodic signal which does not repeat itself. E.g. exponential signals.
- A sinusoidal wave has only one frequency which represents the cycles/sec.

- Other periodic signals might have more than one frequency and f_0 is known as the fundamental frequency which actually represent the cycles/sec for that signal. All other frequencies usually are integer multiple of the fundamental frequency.

2. Random and Deterministic:

- Deterministic signals: are those signals which we can be constructed using a mathematical relation, table lookup or by other means.
- Random signals: those signals which have some level of uncertainty. We can estimate theses signal by statistical and probabilistic means.

3. Power and Energy signals:

- A signal is known as energy signal if it has finite energy but zero average power. For example, deterministic and aperiodic signals.
- A signal is known as power signal if it has finite average power but infinite energy. For example, random and periodic signals.

Simple operations on signals:

1. Scaling $g(t) = mx(t)$ where *m* is a scalar quantity. (Amplification and attenuation)
2. Addition. (E.g. FT concept).
3. Modulation. $g(t) = x(t) \times A\cos 2\pi f_c t$ where $f_0 << f_c$
4. Delay and advancement of a signal. $x(t - t_0)$ and $x(t + t_0)$
5. Time reversal $x(-t)$.

Continuous signals in MATLAB:

- There is no realization of continuous signals in MATLAB.
- A continuous signal is represented by discrete points chosen at distinct points.

Examples:

Sinusoidal signals:

```
A=[10 6 4];          %vary the amplitude. make them equal
p=[0 pi/2 pi];       %vary the phase.
f=40;                %frequency can vary and T accordingly.
n=1;                 %change the cycle number to display.
t=[n*(-0.025):0.001:n*(0.025)]; %choose the range according to the
                                %number of cycles n. You can display
                                %only positive t values.
for k=1:3
    x{k} = A(k).*sin((2*pi*n*f*t)+p(k));
    plot(t,x{k})
    hold on
end

           %Additional plot for all signals with different colors
figure
plot(t,x{1},t,x{2},t,x{3})
clear all
```

Signum function:

```
t=[-10:0.001:10];     %time range. try to increase and decrease range
y = sign(t);          %signum function
plot(t,y);
```

sinc function:

sinc computes the sinc function of an input vector or array, where the sinc function is:

$$\sin c(x) = \begin{cases} 1 & \text{for } t = 0 \\ \dfrac{\sin(\pi x)}{\pi x} & \text{otherwise.} \end{cases}$$

```
A=3;                          %vary
p=40;
t=[(-0.05):0.001:(0.05)];     %only positive t values.

y = A.*sinc(2*p*t);           %sinc function
plot(t,y);

figure
x = abs(A.*sinc(2*p*t));      %magnitude sinc function
plot(t,x);
```

Square wave (single period):

```
A=3;                  %vary
f0=1/10;              %frequency of the square wave
T0=1/f0;              %duration of single pulse
t=[-T0/2:0.01:T0/2];     %range is fixed for one symbol duration


%implementation of square wave.
for k=1:length(t)
    if (abs(t)<=T0)
        x(k)=A;
    else
        x(k)=0;
    end
end

%to connect the last point to zero to give a rectangular shape
%also experiment with the plot properties to see the wave
y=[0,x,0];
t=[-T0/2,t,T0/2];
plot(t,y)
clear all
```

Another way of implementing signals:

- x = sym('x') creates a symbolic object of variable x. So x is actually a symbol variable. Another way to declare a variable to be a symbol object by writing: syms x y So we have two symbols with the name x and y.
- Now x or y can contain any mathematical expression. Try y=x+1
- Implement the following:

```
syms x y z
x
y=z+x+1
x=sym(1/2)
y
y=z+x+1
x=sym(1:5);
y=z+x+1
z=sym(2)
y=z+x+1
```

Impulse function or (Dirac delta):

- Try dirac(2),dirac(-5),dirac(0). This is equivalent to $\delta(t)$.
- Try:

```
t=[-10:0.1:10]; %try decreasing the increment.
x=dirac(t);
plot(t,x)
clear all
```

Integration of a function:

```
syms x y z

y=x+1;
z=int(y,x)       %means integrate y with respect to x
m=int(y,x,0,1)   %means integrate y with respect to x
                 %with the limit from o to 1
clear all
%or try
syms x y z
y=sin(x)
z=int(y,x)
m=int(y,x,-inf,inf)
```

Signal Operations:

```
%The following program is going to perform: Scaling, Addition,
Modulation
%and time shift is left as exercise.

%Scaling
t = [-2:0.001:2];
s = t;
s1 = 5*t;
```

```
plot (t,s)
hold on
plot (t,s1,'r')

x = sin(2*pi*0.5*t);
figure;
plot (t,x)
hold on
plot (t,5*x,'r')
hold on
plot (t,0.2*x,'g')

%Adding

p1 = s1 + x ;
p2 = x + 2;
figure;
plot(t,p1)
hold on
plot(t,p2)

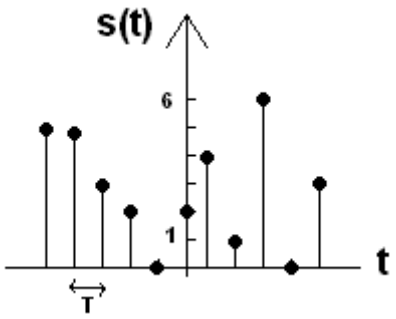
%Modulation

y = 20*cos (2*pi*3*t);
z = s .* y;
figure;
plot(t,z,'m')
```

Exercise:

3.1 Only for positive values of t, if bit 1 is represented by a square wave with +5 Volts and bit 0 with -5 Volts, and if the duration of a bit is 1 second, then plot a time graph for the following sequence: 1011010001

3.2 Implement the following figure using the concept of train of weighted impulses. Hint: integrate the impulse function (i.e. implement $s(t) = \int_{-\infty}^{\infty} \sum_{n=-5}^{5} A_n \delta(t - nT_o) dt$ such that $T_0 = 1$ and A_n is given in the figure below:



Perform the scaling operation of any signal of your own. Show a figure containing the original signal, the delayed signal by some time (t) and the advanced signal by some time (t)

Attach all your graphs here.

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 04

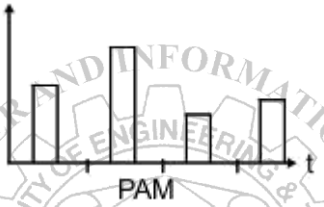
Pulse Amplitude Modulation

OBJECTIVES

1. To get familiar with concept of Pulse Amplitude Modulation
2. To understand concepts of Sampling and Quantization

THEORY

Pulse Amplitude Modulation (PAM) is the simplest form of pulse modulation. This technique transmits data by varying the voltage or power amplitudes of individual pulses in a timed sequence of electromagnetic pulses. In other words, the data to be transmitted is encoded in the amplitude of a series of signal pulses. PAM can also be used for generating additional pulse modulations.



The possible pulse amplitudes in pulse amplitude modulation can be infinite. This is the case with analog pulse amplitude modulation. A 2 level pulse amplitude modulation causes the resulting signal to be digitized while a 4 level pulse amplitude modulation has 22 possible discrete pulse amplitudes. An 8-level pulse amplitude modulation has 23, and 16-level pulse amplitude modulation has 24 discrete pulse amplitudes.

Regarding various pulse amplitude modulation, some systems maintain the amplitude of each pulse directly proportional to the instantaneous modulating-signal amplitude at the time of pulse occurrence. In other pulse amplitude modulation systems, the reverse is true - that is, inversely proportional to the instantaneous modulating-signal amplitude at the time of pulse occurrence. In other pulse amplitude modulation systems, the amplitude is dependent on additional factors related to the modulating signal such as the instantaneous frequency and phase, which may be different than its strength.

While newer technologies are fast making their presence known, it should be noted that pulse amplitude modulation is still useful in the popular Ethernet communication standard. For example, 100BASE-T2 - operating at 100Mbps - Ethernet medium is using 5 level PAM modulations running at 25 mega pulses/sec over four wires. Later developments include the 100BASE-T medium which raised the bar to 4 wire pairs, running each at 125 mega pulses/sec in order to achieve 1000 Mbps data transfer rates, but still with the same PAM5 for each pair.

More recently, PAM12 and PAM8 have gained consideration in the newly proposed IEEE 802.3 a standard for 10GBase-T -- ten gigabyte Ethernet over copper wire.

Two basic steps for Pulse Amplitude Modulation:

- Sampling
- Quantization

Sampling:

An analog signal is characterized by the fact that its attributes (like: amplitude, frequency and phase) can take any value over a continuous range. On the other hand, digital signals can take only discrete and finite values. One can convert an analog signal to a digital signal by sampling and quantizing (collectively called analog-to-digital conversion, or ADC). It is typically more efficient to process the resulting discrete signals by digital signal processors. The processed signals are then converted back into analog signals using a reconstruction or interpolation operation (called digital-to-analog conversion, or DAC).

To sample a continuous-time signal $x(t)$ is to represent $x(t)$ at a discrete number of points, $t = nT_S$ where T_S is the sampling period. The sampling theorem states that a band-limited signal $x(t)$ with bandwidth W can be reconstructed from its sample values $x(n) = x(nT_S)$ if the sampling frequency $f_S = 1/T_S$ is greater than twice the bandwidth W of $x(t)$. Otherwise, aliasing would result in $x(t)$. The minimum sampling rate of $2W$ for an analog band-limited signal is called the Nyquist rate.

Quantization:

In order to process the sampled signal digitally, the sample values have to be quantized to a finite number of levels, and each value can then be represented by a string of bits. For example, if the signal is quantized to N different levels, then $\log_2(N)$ bits per sample are required. Notice that to quantize a sample value is to round it to the nearest point among a finite set of permissible values. Therefore, a distortion will inevitably occur. This is called quantization noise (or error).

Quantization can be classified as uniform and non-uniform. In the case of uniform quantization, the quantization regions are chosen to have equal length. However, in non-uniform quantization, regions of various lengths are allowed. Non-uniform quantization can be implemented through compression-expansion of the signal, and this is commonly used (as in telephony) to maintain a uniform signal-to-quantization noise ratio over the full dynamic range of the signal (refer to your textbook for more details).

EXERCISE

Generate a time vector **t** from -0.5 to 0.5 with a step size of 0.001. Implement the function $x(t) = e^{-|t|/p}$ where *p=0.1*. In order to simulate an A/D converter, perform the following tasks:

4.1 Plot ***x*** and its magnitude spectrum in a two-panel figure window. What is the bandwidth of ***x***?

4.2 Set the sampling frequency to twice the bandwidth of the signal **x** (which is approximately 25Hz). Generate a rectangular pulse train starting at -0.5 to 0.5 where the step size is ***1/ (sampling frequency)*** and with a duration of 0.0001. In a two-panel figure window, plot the pulse train and its magnitude spectrum.

4.3 What is the relation between the time and frequency domain representation of the pulse train?

4.4 Sample ***x*** using the pulse train and plot the resulting sampled version of ***x***. Also, plot its magnitude spectrum. What can you observe from both plots?

Attach your graphs here.

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Lab Session 05

Familiarization with Pulse Code Modulation

OBJECTIVES

1. To get an introductory overview of the variables in PCM coding using MATLAB
2. To give students insight into the relationship between sampling, quantization and signal quality.

THEORY

The primary goal of this lab is to give the student an opportunity to interact with the PCM coding process as shown in figure 5.1. The student will be able to modify the quantization bits to illustrate the quality vs. quantity trade-off in the design of a communications system. This lab should give the student an appreciation and a better understanding of the complexities of digital communications.

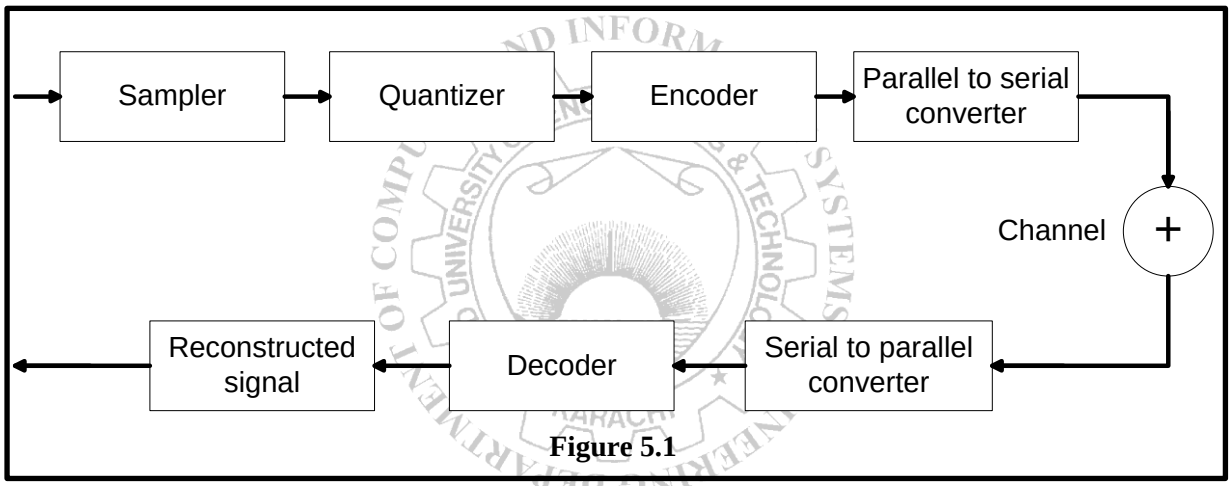


Figure 5.1

The A/D and D/A conversion processes have a number of aspects, but in its simplest form the only thing we need to worry about at this point is that the time spacing (*Ts*) between the signal samples must correspond to the rate of the D-to-A hardware that is being used. A convenient choice for the D-to-A conversion rate is 8000 samples per second, so *Ts* = 1⁄8000 seconds, which is somewhat of a standard in digitizing voice signals. Another common choice is 11,025 Hz which is one-quarter of the rate used for audio CDs. Both of these rates satisfy the requirement of sampling voice signals fast enough (Nyquist rate). For example, CD-quality music is sampled at 44.1 kHz, and DVD quality music is sampled at up to 192 kHz.

In PCM coding, these sample values are converted into bits, which can be stored in a file (or written to a CD), or transmitted. If transmitted, the bits would be fed to a digital modulator and converted to a form suitable for the transmission medium.

Most computers have a built-in analog-to-digital (A/D) converter and a digital-to-analog (D/A) converter (usually on the sound card). In this Lab, we will use pre-recorded audio files to learn more about PCM. These files are collections of bits: an analog music signal has been digitized using PCM.

So using an audio media player and the sound card in your computer these bits can be converted into a voltage waveform that can be used to vibrate the speaker's diaphragm so that we can hear it (a digital-to-analog conversion). These files are ".wav" files; which is one of many standard types of audio files. The file contains enough information describing how the music was recorded (including sample rate and bits/sample) to enable the playback through the sound card and speakers, using a media player program such as the Windows Media Player or MATLAB.

Example:

1. Double-click the MATLAB icon on your desktop and run the MATLAB sound demo by typing **`xpsound`** at the MATLAB command prompt after the program gets started. Choose some of the available sounds in the demo, listen to them, and look at their Time Sequence (or waveform), their Spectrum (frequency content), and their spectrogram (also frequency content, only now a short history of the frequency content over time is displayed).
2. Now close the sound demo window and change your directory in MATLAB. Copy the sound files from desktop. MATLAB knows that a .wav file is a sound file, but we must read in the data in order to process it. To do this, type the following at the MATLAB command line (be sure to type in the single quotes around the file's name and the semi-colon at the end…and don't type the ">>" since it already appears on the command line.

 >> **[a , fs, nbits] = wavread('filename.wav');**

 From the .wav file, variable ***a*** will be the actual digitized voltage waveform that was recorded over microphones and then sampled at *fs* samples/sec, with each sample quantized with *nbits*.
 The sample values in these files all fall somewhere between –1V and +1V (standard for .wav files). If this is CD quality music, the sample rate is 44,100 samples/sec, and there are 16 bits/sample. From the audio file, the value of sample frequency gets stored in variable *fs*, and bits/sample gets stored in variable *nbits*. To see the sample frequency or the bits/sample values, type **fs** or **nbits** at the command line and hit <Enter>.
 From this sample rate and bits/sample, what is the bit rate that developed in ording this music?

 _____bps

3. Play your music clip using the MATLAB *wavplay* command. At the command line, type the following:

 wavplay(a); and hit <Enter>

 This probably doesn't sound too good, as MATLAB will use a default sample rate of 11,025 samples/sec (1/4 of the CD sample rate), unless you tell it otherwise. Speed up the play back by telling MATLAB the correct sample rate:

 wavplay(a, fs); and hit <Enter>

 Now try a higher value of *fs*: tell MATLAB that the sample rate was 1.5**fs* Type:

 wavplay(a , 1.5*fs); and hit <Enter>:

This sounds more like the Chipmunks, and illustrates the importance of knowing the sample rate in an audio (or video) playback application: in order to recover the original signal as it was intended to be heard (or seen), the device that is performing the playback must know the proper interval (in seconds) between samples

With the given quality sample rate, what is the time interval between samples?

_____ Sec

4. Using the resolution, it is easy to figure out the maximum quantization error that could occur in the quantization process (max quantizing error = ½ resolution), and to figure out the Signal-to-Quantization Noise ratio in dB (*SQRdB* = 1.8 + 6*n*, where *n* = bits/sample). Fill out the following table for your chosen audio clip. Each time you change the quantization, use the

soundsc function as you did before to listen to the degraded result:
soundsc(a , fs, n); and hit <Enter>:

Bits/sample	**Resolution (volts)**	**Max Quantizing Error(volts)**	**Signal-to-Quantization Noise, SQRdb**
14			
8			
5			
3			

Exercise:

Record your voice in a short 15-25 seconds sample

Step1: Read in the wav file using wavread()

Step2: Listen to your voice using sound()

Step3: Listen to your voice at twice and half the sampled frequency (2*f and 0.5*f), what happens when you divide 'f' by a factor say '1/2*f'?

Step4: Listen to it at twice and half the amplitude (2*x and 0.5*x), what happens when you multiply 'x' by a factor say '2*x'?

Step5:

Flip the array 'x' using flipud(x) and listen to it. Make sure you save the flipped array in a new variable. It should play the sound backwards. Repeat steps 3 and 4 with this backward sample.

For example:

>>Q=flipud(x)

>>sound(Q,fs)

Attach your graphs here.

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
KARACHI

Section Two

Modulation and Demodulation Techniques in Bandpass

Lab Session 06

Introduction to Modulation techniques

OBJECTIVES

1. Getting Familiar with Bandpass modulation.
2. Analyzing all major types of modulation techniques.

THEORY

DIGITAL BANDPASS MODULATION TECHNIQUES

Bandpass modulation (either analog or digital) is the process by which an information signal is converted to a sinusoidal waveform; for digital modulation, such a sinusoid of duration *T* is referred to as a digital symbol. The sinusoid has just three features that can be used to distinguish it from other sinusoids: amplitude, frequency, and phase. Thus bandpass modulation can be defined as the process whereby the amplitude, frequency, or phase of an RF carrier, or a combination of them, is varied in accordance with the information to be transmitted. The general form of the carrier wave is

$$s(t) = A(t)\ \cos\theta(t) \tag{6.1}$$

Where $A(t)$ is the time-varying amplitude and θ(t) is the time-varying angle. It is convenient to write.

$$\theta(t) = \omega_0 t + \emptyset(t) \tag{6.2}$$

So that

$$s(t) = A(t)\ \cos\ [\omega_0\ t + \emptyset(t)\] \tag{6.3}$$

Where ω_0 is the radian frequency of the carrier and $\emptyset(t)$ is the phase. The terms *f* and ω will each be used to denote frequency. When *f* is used, frequency in hertz is intended; when *w* is used. Frequency in radians per second is intended. The two frequency parameters are related by w=2πf.

The basic bandpass modulation/ demodulation types are listed in Figure 6.1. When the receiver exploits knowledge of the carrier's phase to detect the signals the process is called coherent detection. When the receiver does not utilize such phase reference information the process is called noncoherent detection. In digital communication, the terms demodulation and detection are often used interchangeably, although demodulation emphasized waveform recovery, and detection entails the process of symbol decision. In ideal coherent detection, there is available at the receiver a prototype of each possible arriving signal. These prototype waveforms attempt to duplicate the transmitted signal set in every respect, even RF phase. The receiver is then said to be phase locked to the incoming signal. During demodulation, the receiver multiplies and integrates (correlates the incoming signal with each of its prototype replicas. Under the heading of coherent modulation/ demodulation in Figure 6.1 are listed phase shift keying (PSK),

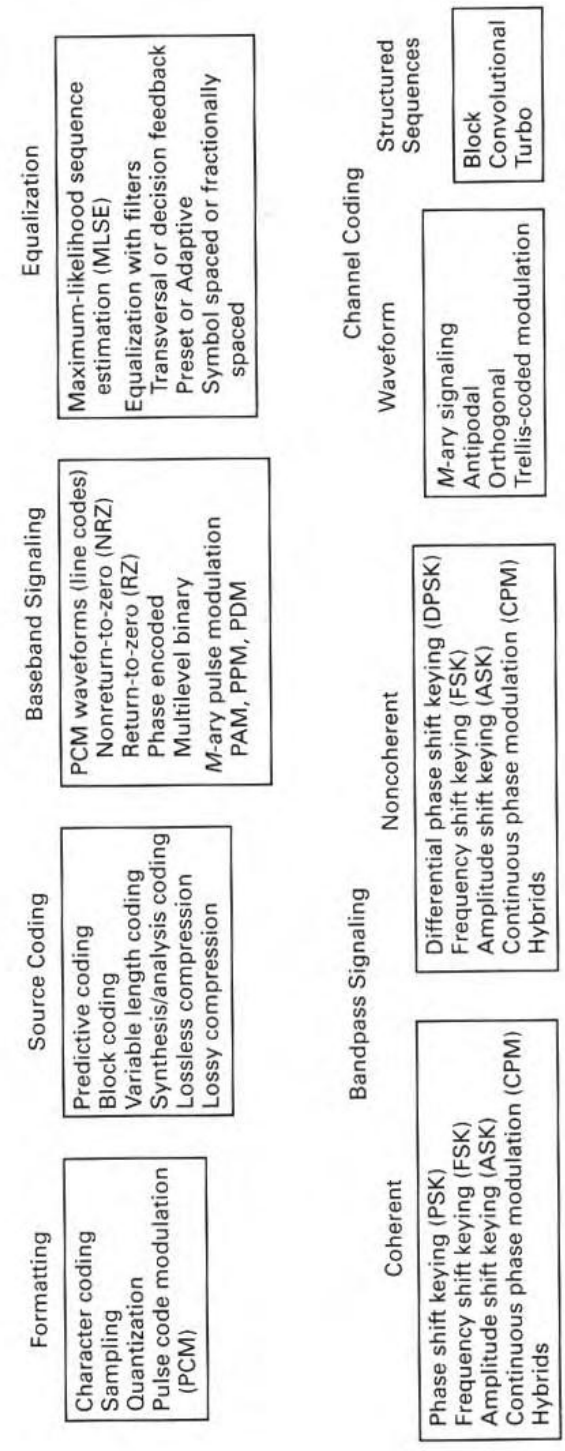


Figure 6.1 Basic Digital communication transformations

Page No. 9

Book : DIGITAL COMMUNICATIONS Fundamentals and Applications
Second Edition by BERNARD SKLAR

frequency shift keying (FSK), amplitude shift keying (ASK), continuous phase modulation (CPM), and hybrid combination. The basic bandpass modulation formats are discussed in this lab. Some specialized formats, such as offset quadrature PSK (OQPSK), minimum shift keying (MSK) belonging to the CPM class, and quadrature amplitude modulation (QAM), are treated in your theory course.

Noncoherent demodulation refers to systems employing demodulators that are designed to operate without knowledge of the absolute value of the incoming signal's phase; therefore, phase estimation is not required. Thus the advantage of noncoherent over coherent systems is reduced complexity, and the price paid is increased probability of error (P_E). In figure 6.1 the modulation/demodulation types that are listed in the noncoherent column, DPSK, FSK, ASK, CPM and hybrids, are similar to those listed in the coherent column. We had implied that phase information is not used for noncoherent reception; how do you account for the fact that there is a form of phase shift keying under the noncoherent heading? It turns out that an important form of PSK can be classified as noncoherent (or differentially coherent) since it does not require a reference in phase with the received carrier. This "pseudo-PKS," termed differential PSK (DPSK), utilizes phase information of the prior symbol as a phase reference for detecting the current symbol.

Phase-shift keying

Phase-shift keying (PSK**)** is a digital modulation scheme that conveys data by changing, or modulating, the phase of a reference signal (the carrier wave).

Any digital modulation scheme uses a finite number of distinct signals to represent digital data. PSK uses a finite number of phases; each assigned a unique pattern of binary digits. Usually, each phase encodes an equal number of bits. Each pattern of bits forms the symbol that is represented by the particular phase. The demodulator, which is designed specifically for the symbol-set used by the modulator, determines the phase of the received signal and maps it back to the symbol it represents, thus recovering the original data. This requires the receiver to be able to compare the phase of the received signal to a reference signal — such a system is termed coherent (and referred to as CPSK).

Alternatively, instead of using the bit patterns to *set* the phase of the wave, it can instead be used to *change* it by a specified amount. The demodulator then determines the *changes* in the phase of the received signal rather than the phase itself. Since this scheme depends on the difference between successive phases, it is termed **differential phase-shift keying (DPSK)**. DPSK can be significantly simpler to implement than ordinary PSK since there is no need for the demodulator to have a copy of the reference signal to determine the exact phase of the received signal (it is a non-coherent scheme). In exchange, it produces more erroneous demodulations. The exact requirements of the particular scenario under consideration determine which scheme is used.

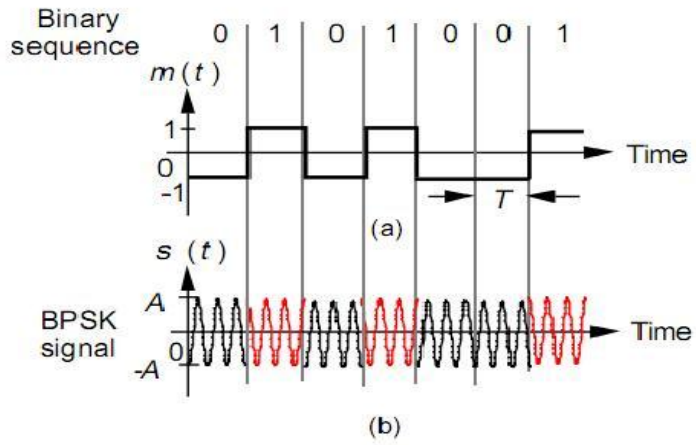
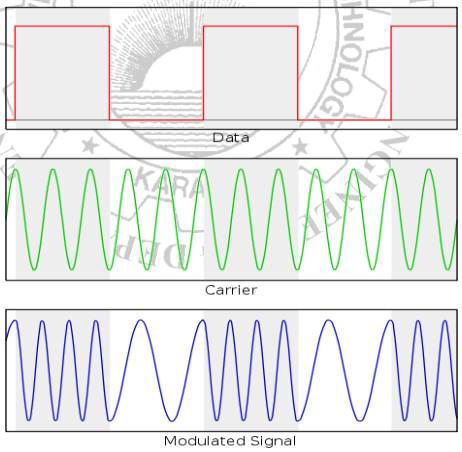


Figure 6.2 (a) Binary modulating signal, and (b) BPSK signal

Frequency-shift keying

Frequency-shift keying (FSK) is a frequency modulation scheme in which digital information is transmitted through discrete frequency changes of a carrier wave. The simplest FSK is binary FSK (BFSK). BFSK literally implies using a pair of discrete frequencies to transmit binary (0s and 1s) information. With this scheme, the "1" is called the mark frequency and the "0" is called the space frequency. The time domain of an FSK modulated carrier is illustrated in the figure given below.



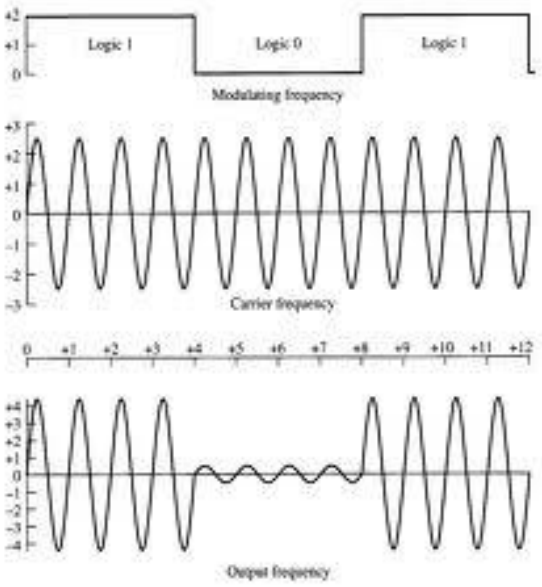
Amplitude-Shift Keying:

Amplitude-shift keying (ASK) is a form of modulation that represents digital data as variations in the amplitude of a carrier wave.

The amplitude of an analog carrier signal varies in accordance with the bit stream (modulating signal), keeping frequency and phase constant. The level of amplitude can be used to represent binary logic 0s and 1s. We can think of a carrier signal as an ON or OFF switch. In the modulated signal, logic 0 is represented by the absence of a carrier, thus giving OFF/ON keying operation and hence the name given.

The ASK technique is also commonly used to transmit digital data over optical fiber. For LED transmitters, binary 1 is represented by a short pulse of light and binary 0 by the absence of light.

Laser transmitters normally have a fixed "bias" current that causes the device to emit a low light level. This low level represents binary 0, while a higher-amplitude light wave represents binary 1.



EXERCISE

Generation of bandpass modulated signals

6.1 Amplitude-Shift Keying.

Generate a binary sequence with the first 6 bits, and set the binary data rate `Fd` to 1 kb/s and the simulation sampling rate `Fs` to 50 kHz:

```
>> global Fd Fs;
>> Fd = 1000;
>> Fs = 50000;
>> b = [1 0 1 0 0 1 round(rand(1,994))];
```

If using the student edition of MATLAB, limit the bit sequence to 250 bits. However, this will result in hard to identify power spectra.

Modulate this sequence into an ASK signal with a carrier frequency of 5 kHz by first generating a unipolar NRZ signal s from the sequence b, then mixing this sequence with a local oscillator with a frequency of 5 kHz.

```
>> s = modul(b,'unipolar_nrz');
>> sa = mix_lo(s,5000,0);
```

Display the first 6 ms of the signals s and sa corresponding to the first 6 bits of the binary sequence b. Examine the relationship between the two signals.

```
>> clf, subplot(211), scope(s,'one-shot',0.006);
>> subplot(212), scope(sa,'one-shot',0.006);
```

6.1.1 **Plot the PSD functions of s and sa between 0 and 10 kHz:**

```
>> [Ss,f] = psd(s,2500,Fs,'mean');
>> clf, subplot(211), plot(f(1:500),Ss(1:500));
>> title('Graph 1-a: PSD of unipolar NRZ');
>> [Ssa,f] = psd(sa,2500,Fs,'mean');
>> subplot(212), plot(f(1:500),Ssa(1:500));
>> title('Graph 1-b: PSD of ASK');
```

6.2 **Phase-Shift Keying.**

Generate a PSK signal by first modulating `b` into a bipolar NRZ signal `s` and then mixing `s` with a 5 kHz local oscillator:

```
>> s = modul(b,'bipolar_nrz');
>> sp = mix_lo(s,5000,0);
```

Display the first 6 ms of the waveforms `s` and `sp`:

```
>> clf, subplot(211), scope(s,'one-shot',0.006);
>> subplot(212), scope(sp,'one-shot',0.006);
```

What is the phase difference between `sp` and the carrier during the first and second bit periods?

6.2.1 **Now plot the PSD functions of `s` and `sp`:**

```
>> [Ss,f] = psd(s,2500,Fs,'mean');
>> clf, subplot(211), plot(f(1:500),Ss(1:500));
>> [Ssp,f] = psd(sp,2500,Fs,'mean');
>> subplot(212), plot(f(1:500),Ssp(1:500));
```

Do you observe any relationship between the PSD of `s` and that of `sp`?

We now compare the power spectrum of ASK with that of PSK.

```
>> clf, subplot(211), plot(f(1:500),Ssa(1:500));
>> title('Graph 2-a: PSD of ASK');
>> subplot(212), plot(f(1:500),Ssp(1:500));
>> title('Graph 2-b: PSD of PSK');
```

Zoom in on the two power spectra so as to have 'enough' detail and about the same scaling for both. What differences and/or similarities do you notice?

6.3 **Frequency-Shift Keying.**

Generation of a FSK signal also starts with a bipolar NRZ signal (`s` from the previous section) which is applied to the input of a voltage-controlled oscillator (VCO) with a free-running frequency of 5 kHz and a sensitivity of 2 kHz/V.

```
>> sf = vco(s,5000);
```

Display the first 6 ms of the waveforms `s` and `sf`:

```
>> clf, subplot(211), scope(s,'one-shot',0.006);
>> subplot(212), scope(sf,'one-shot',0.006);
```

6.3.1 **Display the PSD function of `sf`:**

```
>> [Ssf,f] = psd(sf,2500,Fs,'mean');
>> clf, plot(f(1:500),Ssf(1:500));
>> title('Graph 3: PSD of FSK');
```

Explain the shape of the obtained power spectrum.

Attach all your graphs here.

Lab Session 07

Understanding Demodulation techniques

OBJECTIVES

1. To understand the basic concept of demodulation and problems associated with it.
2. To familiarize with coherent detection of bandpass modulated digital signal.
3. To familiarize with non-coherent detection of bandpass modulated digital signal.

THEORY

Demodulation is the act of extracting the original information-bearing signal from a modulated carrier wave. A demodulator is an electronic circuit (or computer program in a software defined radio) that is used to recover the information content from the modulated carrier wave.
These terms are traditionally used in connection with radio receivers, but many other systems use many kinds of demodulators. Another common one is in a modem, which is a contraction of the terms modulator/demodulator.

There are several ways of demodulation depending on how parameters of the base-band signal are transmitted in the carrier signal, such as amplitude, frequency or phase. For example, for a signal modulated with a linear modulation, like AM (Amplitude Modulated), we can use a synchronous detector. On the other hand, for a signal modulated with an angular modulation, we must use an FM (Frequency Modulation) demodulator or a PM (Phase Modulation) demodulator. Different kinds of circuits perform these functions.

Many techniques—such as carrier recovery, clock recovery, bit slip, frame synchronization, rake receiver, pulse compression, Received Signal Strength Indication, error detection and correction, etc. -- are only performed by demodulators, although any specific demodulator may perform only some or none of these technique

Coherent detection of bandpass modulated digital signals:

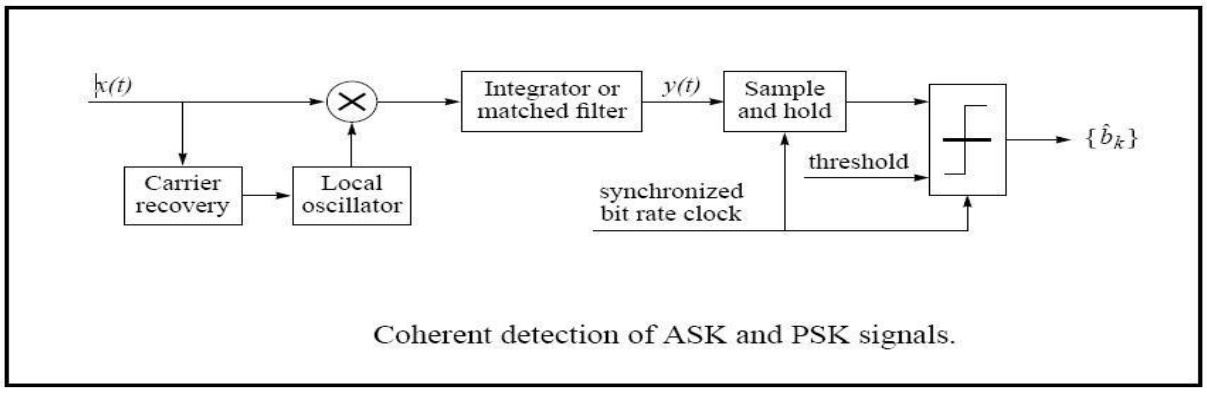


Figure 7.1 illustrates the structure of a coherent detector applicable to ASK and PSK signals.

Practice Problem 7.1.1:

To demodulate the ASK signal `sa`, start by mixing it with a carrier produced by a local oscillator with the same frequency and phase as the original LO. Display the resulting waveform `xa` for the first six bit periods (6 ms). Also, produce and display the PSD function of `xa` over the interval.

```
>> xa = mix_lo(sa,5000,0);
>> clf, subplot(211), scope(xa,'one-shot',0.006);
>> [Sxa,f] = psd(xa,2500,Fs,'mean');
>> subplot(212), plot(f(1:750),Sxa(1:750));
```

Practice Problem7.1.2:

Apply `xa` to the input of a unipolar NRZ matched filter and display the output waveform over the first 6 bit periods.

```
>> ya = afilter(xa,matched('unipolar_nrz'));
>> subplot(212), scope(ya,'one-shot',0.006);
```

Practice Problem 7.2:

We will now examine the performance of coherent detection in the presence of noise. Users of MATLAB student edition should set the `MATLAB_SE` global variable:

```
>> global MATLAB_SE; MATLAB_SE = 1;
```

First, generate a 10000-bit binary sequence and modulate it as an ASK waveform:

```
>> Fs = 20000;
>> b = round(rand(1,10000));
>> s = mix_lo(modul(b,'unipolar_nrz'),5000,0);
```

Transmit `sa` through a channel with additive noise of power of 0.5 W:

```
>> x = bbchannel(s,1,0.5,8000);
```

Using coherent detection principles, demodulate `x` and display the eye pattern over the duration of 200 bits2 3.

```
>> y = afilter(mix_lo(x,5000,0),matched('unipolar_nrz'));
>> clf, scope(y(1:4000),'continuous',0.002);
```

Find the optimal sampling instant and threshold level and proceed to compute the BER:

```
>> detect(y,<threshold>,<sampling_time>,b)
```

The main difficulty in implementing coherent detection lies in the need to have perfectly synchronized local oscillators at both the transmitter and the receiver. Various methods can be used for this purpose that will not be explained here. The result, however, is that no practical system using coherent detection is completely free of the impediments called *phase jitter* and *frequency drift*.

Practice Problem 7.3:

Phase error. Repeat the experiment above with a PSK waveform for 1 W of noise power and with the addition of a phase error in the demodulation process, e.g.:

```
>> y = afilter(mix_lo(x,5000,<phase_error>),
matched('bipolar_nrz'));
```

where the phase error is specified in degrees. Since the case has already been treated in the previous section.

Frequency drift. Generate a 500-bit binary sequence and modulate it to produce an ASK waveform:

```
>> b = round(rand(1,500));
```

This particular eye pattern might be closed and difficult to interpret. Try plotting it for a shorter portion of the sequence, say of 100 bits. You can estimate where the opening should be by the shape of the signal lines. Locate the threshold as being in the middle of the pattern, i.e. take the average of the maximum and minimum observable voltage levels. Remember that for bipolar schemes, the optimal threshold should always be zero. In order to find you may need to resort to an iterative trial and error procedure by searching for the minimum obtainable BER. Start with a value of 0 ms and increment it by 0.1 ms until the resulting BER stops decreasing.

Practice problem 7.4:

```
>> s = mix_lo(modul(b,'unipolar_nrz'),5000,0);
```

Demodulate this signal using a perfectly frequency-locked L.O. (5 kHz) and using a L.O. suffering a slight frequency drift (4.975 kHz):

```
>> y = afilter(mix_lo(s,5000,0),matched('unipolar_nrz'));
>> yd = afilter(mix_lo(s,4975,0),matched('unipolar_nrz'));
```

Observe and compare the two waveforms over the first 50 bit periods:

```
>> clf, subplot(211), scope(y,'one-shot',0.05);
>> subplot(212), scope(yd,'one-shot',0.05);
```

Practice problem 7.5:

Visually determine the frequency of the envelope of the matched filter output `yd`.
You will now modulate the sequence `b` you generated above to produce a PSK waveform:

```
>> s = mix_lo(modul(b,'bipolar_nrz'),5000,0);
```

Demodulate `s` as you did above for the ASK case using the following local oscillator frequencies: {4999.5, 4999.4, 4999.3, 4999.2, 4999.1, 4999.0} Hz. Remember to use a matched filter corresponding to the bipolar NRZ pulse. Find the bit error rate for each case. You may want to use the eye pattern to determine the optimum sampling instant and threshold.

Noncoherent detection of bandpass modulated digital signals

Figure 7.2 illustrates the structure of a noncoherent receiver for ASK signals. Noncoherent detection does not require a local oscillator synchronized with the carrier. However, the obtained with a noncoherent receiver is generally higher than that obtained with a coherent receiver with good oscillator lock.

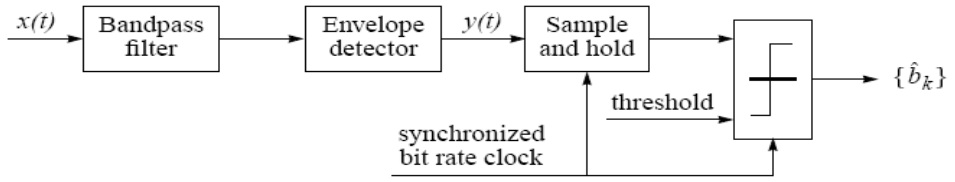


Fig 7.2: Noncoherent detection of ASK signals using an envelope detector.

The bandpass filter typically has a bandwidth of and serves to limit the out-of-band noise and interference in the system. The envelope detector consists of a rectifier followed by a low-pass filter with appropriate cut-off frequency. It is important to consider the bandwidth of the signal and the carrier frequency, since its role is to preserve signal information while eliminating the rapid fluctuations due to the carrier.

To illustrate, create an ASK waveform and demodulate it using a noncoherent receiver whose envelope detector has a cut-off frequency of 3500 Hz.

```
>> b = [1 0 1 0 0 1 1 0 round(rand(1,12))];
>> sa = mix_lo(modul(b,'unipolar_nrz'),5000,0);
>> ya = envelope(sa,3500);
```

Display the first 8 bit periods of both the modulated waveform and the output of the envelope detector. Can threshold decoding be applied at the output of the envelope detector?

Repeat the ASK portion of above mention section with a noise power of 0.05 W for both coherent and noncoherent detection. For noncoherent detection you have to replace the demodulation and matched filtering with envelope detection. Use a 10000 bit sequence as your starting point:

```
>> b = round(rand(1,10000));
```

Note that this is not the optimum noncoherent receiver but only a simplified implementation. As such, this receiver structure does not optimize the signal to noise ratio.

For coherent detection use the threshold and value determined in *practice problem 7.2*, while for noncoherent detection you will have to display the corresponding eye pattern.

EXERCISE:

7.1 Explain the result obtained in practice problem 7.1.1 and also attach its graph at the last of the lab session.

7.2 Explain the impact of the result of practice problem 7.1.2 on the results obtained from practice problem 7.1.1

7.3. Record your results obtained from practice problem 7.2.1 in Table 7.1 then repeat the above for a noise power of 1 W. Repeat the experiment using a PSK waveform. To generate and demodulate a PSK waveform, use the same procedure as for ASK but substitute `'bipolar_nrz'` instead of `'unipolar_nrz'` as the argument of `modul` and `matched`. Display the eye pattern again to find the optimum sampling time and threshold for the PSK case.

Table 7.1: ASK and PSK bit error rate

Noise power	P_e – ASK	P_e – PSK
0.5 W		
1.0 W		

7.4 For problem discussed in Practice Problem 7.3. Write down the Phase error calculated for frequencies given in Table 7.2

Table 7.2 Phase error

LO frequency [Hz]	4999.5	4999.4	4999.3	4999.2	4999.1	4999.0
P_e						

7.5 Explain the results obtained in Practice Problem 7.4 and also attach their graphs at the end of the lab session.

7.6 List down the result in Table 7.3 obtained from Practice Problem 7.5

Table 7.3 PSK BER vs. frequency error

LO frequency [Hz]	4999.5	4999.4	4999.3	4999.2	4999.1	4999.0
P_e						

Attach your graphs here

Section Three

Channel Coding

Lab Session 08

Encoding Linear Block Code

OBJECTIVES

1. To understand fundamentals of Linear Block Codes
2. Familiarization with Linear Encoding Techniques.

THEORY

In mathematics and information theory, **linear code** is an important type of block code used in error correction and detection schemes. Linear codes allow for more efficient encoding and decoding algorithms than other codes.

Linear codes are applied in methods of transmitting symbols (e.g., bits) on communications channel so that, if errors occur in the communication, some errors can be detected by the recipient of the message block. A linear code of length n transmits blocks containing n symbols. For example, the (7, 4) Hamming code is a binary linear code which represents 4-bit values with 7 bits. In this way, the recipient can detect errors as severe as 2 bits per block.

In linear block codes, a block of k information bits is followed by a group of r check bits derived from the information bits and at the receiver the check bits are used to verify the information bits which are preceding the check bits.

No of code words = 2^k

Block length = n

Code rate = k/n

Each block of k bits is encoded into block of n bits (n>k) by adding n-k = r check bits.

Where

$$\mathbf{2^r \geq k + r + 1} \qquad \textbf{(8.1)}$$

The check bits are determined by some predetermined rule.

$$\mathbf{C = D\ G}$$

Where:
C = code vector

D = Data (message) vector

G = Generator matrix which is defined as [I_k | P]; I_k is identity matrix of order k and P is the predefined encoding rule.

MATLAB Syntax:

The encode function is used for encoding. The syntax is as follows.

code = encode(msg,n,k,'linear/fmt',genmat)

where, the codeword length is n and the message length is k.

msg represents data or message. It can be in decimal or binary format. The default value for this parameter is binary. We will use binary format in this and forth coming lab sessions.

For example:

Format of msg can be a binary column vector as given below.

msg = [0 1 1 0 0 1 0 1 1 0 0 1]'. The ' symbol indicates matrix transpose.

Format of msg can also be binary matrix with k columns. In this case format of code will be binary matrix with n columns.

msg = [0 1 1 0; 0 1 0 1; 1 0 0 1]. Here k = 4.

For linear block codes, encode function encodes msg using genmat as the generator matrix. genmat, a k-by-n matrix, is required as input.

Example:

The example below illustrates two different information formats (binary vector and binary matrix) for linear block code. The two messages have identical content in different formats. As a result, the two codes created by encode function have identical content in correspondingly different formats.

Here k = 11. Putting r = 4 to satisfy (**Eq. 8.1**)

```
r = 4;                              % r is the number of check bits.
k = 11;                             % Message length
n = k + r                           % Codeword length = 15 using formula n-k=r
                                    % Create 100 messages, k bits each.
msg1 = randint(100*k,1,[0,1]);      % As a column vector
msg2 = vec2mat(msg1,k);             % As a k-column matrix
                                    % Create 100 codewords, n bits each.
   P =[1 1 1 1;  0 1 1 1;  1 1 1 0;  1 1 0 1;  0 0 1 1;  0 1 0 1;  0 1 1 0;  1 0 0 1;  1 0 1 0;
       1 0 1 1;  1 1 0 0]
genmat=[eye(11) P]; % concatenate P submatrix or predefined rule with Identity matrix.
code1 = encode(msg1,n,k,'linear/binary',genmat);
code2 = encode(msg2,n,k,'linear/binary',genmat);
if ( vec2mat(code1,n)==code2 )
  disp('All two formats produced the same content.')
end
```

Instead of randomly generating data words, you can create a data matrix of your own containing the data words that you want to encode.

EXERCISE

8.1 Given (6,3) linear block code generated by the predefined matrix [0 1 1; 1 0 1; 1 1 0].

a) Encode the messages [1 1 1] and [1 0 1] manually and verify through MATLAB.

111 = _____. 101 = _____

NED UNIVERSITY OF ENGINEERING & TECHNOLOGY
DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING
KARACHI

b) Encode all the possible messages in MATLAB and write the encoded words.

Message	Code Vector
000	
001	
010	
011	
100	
101	
110	
111	

c) The minimum hamming distance = _____.

d) The no. of errors this code can detect is _____

e) The no. of errors this code can correct is _____

f) The code rate = _____.

g) Is this code a systematic code? Justify your answer. (HINT: see the tabulated code words obtained in 8.1.b)

8.2 Consider a systematic (6, 3) block code generated by the submatrix [1 1 0; 0 1 1; 1 0 1].

a) Write the value of n and k.

b) Tabulate the code words through simulation and write their corresponding weights.

Message	Code Vector	Weight
000		
001		
010		
011		
100		
101		
110		
111		

c) What is the value of d_{min} ?

8.3 Write MATLAB code to create 10 messages randomly having size of 4 bits each. Design appropriate encoding rule and use `encode` function to encode the generated messages.

Lab Session 9

Decoding Linear Block Codes

OBJECTIVE

Familiarization with decoding techniques of Linear Block Codes.

THEORY

Let Y stands for the received vector when a particular code vector X has been transmitted. Any transmission errors will result in $Y \neq X$. The decoder detects or corrects errors in Y using stored information about the code.

A direct way of performing error detection would be to compare Y with every vector in the code. This method requires storing all 2^k code vectors at the receiver and performing up to 2^k comparisons. But efficient codes generally have large values of k, which implies rather extensive and expensive decoding hardware.

More practical decoding methods for codes with large k involve parity check information derived from the code's P submatrix. Associated with any systematic linear (n, k) block code is a $(n-k) \times n$ matrix called the **parity check matrix H**. This matrix is defined by

$$\mathbf{H = [P' \mid I_r]}$$

Where I_r is the $r \times r$ identity matrix and $n - k = r$. The parity check matrix has a crucial property for error detection which is

$$\mathbf{XH^T = (0\ 0\ 0\ 0\ \ldots\ldots 0)}$$

provided that X belongs to the set of code vectors. However, when Y is not a code vector, the product YH^T contains at least one nonzero element.

Therefore, given H^T and a received vector Y, error detection can be based on

$$\mathbf{S = Y\,H^T}$$

an r-bit vector called the **syndrome**. If all elements of S equal zero, then either Y equals the transmitted vector X and there are no transmission errors, or Y equals some other code vector and the transmission errors are undetectable. Otherwise, errors are indicated by the presence of nonzero elements in S. Thus a decoder for error detection simply takes the form of a syndrome calculator.

We develop the decoding method by introducing an n-bit error vector E whose non zero elements mark the positions of transmission errors in Y. For instance, if X = (1 0 1 1 0) and Y = (1 0 0 1 1) then E = (0 0 1 0 1).

In general,

$$\mathbf{Y = X \oplus E}$$

And conversely,

$$\mathbf{X = Y \oplus E}$$

Substituting Y = X + E into $S = YH^T$, we obtain

$$\mathbf{S = (X \oplus E)H^T}$$

$$\mathbf{S = XH^T \oplus EH^T}$$

$$\mathbf{S = EH^T}$$

which reveals that the syndrome depends entirely on the error pattern, not the specific transmitted vector.

Syndrome decoding: example

An (8, 4) binary linear block code C is defined by systematic matrices:

$$G = \left(\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right) \qquad H = \left(\begin{array}{cccc|cccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

Consider two possible messages:

m1 = [0 1 1 0] m2 = [1 0 1 1]

c1 = [0 1 1 0 0 1 1 0] c2 = [0 1 0 0 1 0 1 1]

Suppose error pattern e = [0 0 0 0 0 1 0 0] is added to both code words.

r1 = [0 1 1 0 0 0 1 0] r2 = [0 1 0 0 1 1 1 1]

Calculating the syndrome using $S = YH^T$. Here Y is the received erroneous code word.

s1 = [0 1 0 0] s2 = [0 1 0 0]

The syndromes are the same and equal to the 6th column of H, so decoder corrects bit 6.

Bit Error Rate:

Bit error rate = average no. of erroneous bits per block / total no. of bits per block

MATLAB FUNCTIONS:

In order to develop MATLAB code for decoding and calculating the bit error rate, the following functions may be used.

1) DECODE:

Function: Block decoder

Syntax

```
msg = decode(code,n,k,'linear/fmt',genmat,trt)
[msg,err] = decode(...)
[msg,err,ccode] = decode(...)
[msg,err,ccode,cerr] = decode(...)
```

`trt` Uses `syndtable` to create the syndrome decoding table associated with the method's parity-check matrix. `Syndtable` function is described later in this lab session.

Description

The `decode` function aims to recover messages that were encoded using an error-correction coding technique. The technique and the defining parameters must match those that were used to encode the original signal.

Lab session #8 explains the meanings of `n` and `k`, the possible values of `fmt`, and the possible formats for `msg` and `code` for the `encode` function. You should be familiar with the conventions described there. Using the `decode` function with an input argument `code` that was not created by the `encode` function might cause errors.

`Decode` function decodes code, which is a linear block code determined by the `k`-by-`n` generator matrix `genmat`. `genmat` is required as input. `decode` tries to correct errors using the decoding table `trt`, where `trt` is a `2^(n-k)`-by-`n` matrix.

`[msg,err] = decode(...)` returns a column vector `err` that gives information about error correction. A nonnegative integer in the rth row of `err` (or the r^{th} row of `vec2mat(err,k)` if `code` is a column vector) indicates the number of errors corrected in the r^{th} message word. A negative integer indicates that there are more errors in the r^{th} word than can be corrected.

`[msg,err,ccode] = decode(...)` returns the corrected code in `ccode`.

`[msg,err,ccode,cerr] = decode(...)` returns a column vector `cerr` whose meaning depends on the format of code:

- If `code` is a binary vector, then a nonnegative integer in the r^{th} row of `vec2mat(cerr,n)` indicates the number of errors corrected in the r^{th} codeword. A negative integer indicates that there are more errors in the r^{th} codeword than can be corrected.
- If `code` is not a binary vector, then `cerr = err`.

2) SYNDTABLE:

Function: Produces syndrome decoding table

Syntax:

```
t = syndtable(h)
```

Description

`t = syndtable(h)` returns a decoding table for an error-correcting binary code having codeword length `n` and message length `k`. `h` is an `(n-k)`-by-`n` parity-check matrix for the `code`. `t` is a 2^{n-k}-by-`n` binary matrix. The r^{th} row of `t` is an error pattern for a received binary codeword whose syndrome has decimal integer value `r-1`. (The syndrome of a received codeword is its product with the transpose of the parity-check matrix.).

3) BITERR:

Function: Computes number of bit errors and bit error rate.

Syntax

```
[number,ratio] = biterr(x,y)
```

Description

The biterr function compares unsigned binary representations of elements in vector `x` with those in vector `y`.

Each element of `x` and `y` must be a nonnegative decimal integer; `biterr` converts each element into its natural unsigned binary representation. `number` is a scalar or vector that indicates the number of bits that differ. `ratio` is `number` divided by the total number of bits. The total number of bits, the size of number, and the elements that `biterr` compares are determined by the dimensions of `x` and `y`.

`Biterr` function compares the elements in `x` and `y`. If the largest among all elements of `x` and `y` has exactly `k` bits in its simplest binary representation, then the total number of bits is `k` times the number of entries in the smaller input. The sizes of `x` and `y` determines which elements are compared:

- If `x` and `y` are matrices of the same dimensions, then `biterr` compares `x` and `y` element by element, number is a scalar. See fig 9.1.

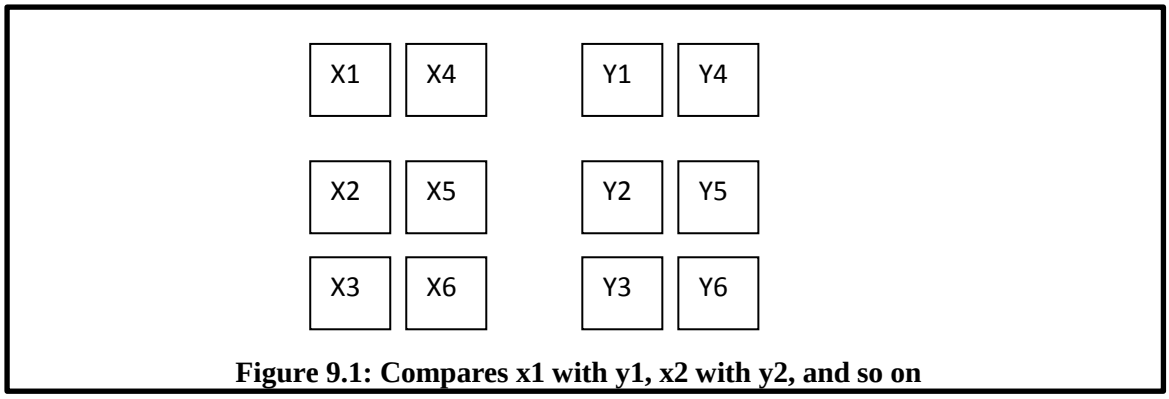
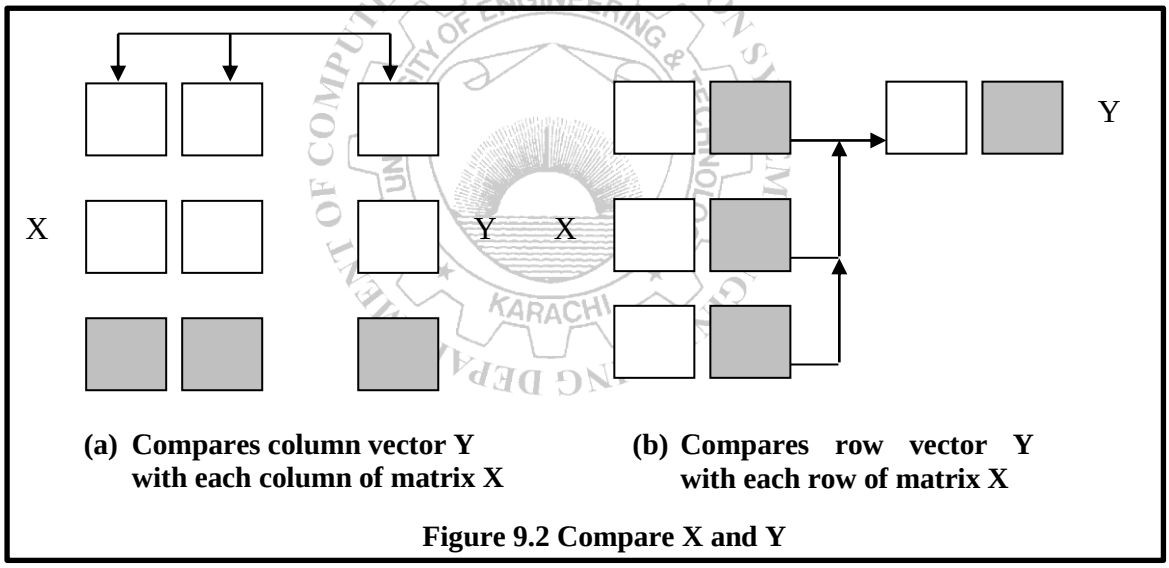


Figure 9.1: Compares x1 with y1, x2 with y2, and so on

- If one is a row (respectively, column) vector and the other is a two-dimensional matrix, then biterr compares the vector element by element with each row (resp., column) of the matrix. The length of the vector must equal the number of columns (resp., rows) in the matrix. Number is a column (resp., row) vector whose m^{th} entry indicates the number of bits that differ when comparing the vector with the m^{th} row (resp., column) of the matrix. See schematics in figure 9.2.



(a) Compares column vector Y with each column of matrix X

(b) Compares row vector Y with each row of matrix X

Figure 9.2 Compare X and Y

4) gen2par

Function: Convert between parity-check and generator matrices

Syntax

```
parmat = gen2par(genmat)
genmat = gen2par(parmat)
```

Description

`parmat = gen2par(genmat)` converts the standard-form binary generator matrix `genmat` into the corresponding parity-check matrix `parmat`.

`genmat = gen2par(parmat)` converts the standard-form binary parity-check matrix `parmat` into the corresponding generator matrix `genmat`.

The standard forms of the generator and parity-check matrices for an `[n,k]` binary linear block code are shown in the table below.

Type of Matrix	Standard Form	Dimensions
Generator	$[I_k \mid P]$ or $[P \mid I_k]$	`k`-by-`n`
Parity-check	$[P' \mid I_{n-k}]$ or $[I_{n-k} \mid P']$	`(n-k)`-by-`n`

where I_k is the identity matrix of size `k` and the ' symbol indicates matrix transpose. Two standard forms are listed for each type, because different authors use different conventions.

Example 1

The command below compares the column vector [0; 0; 0] to each column of a random binary matrix. The output is the number and proportion of 1s in the matrix.

```
number,ratio] = biterr([0;0;0],randint(3,5))

The output is

number =

     2     1     1     0     1

ratio =

    0.6667    0.3333    0.3333         0    0.3333
```

Example 2

The script below adds errors to 10% of the elements in a matrix. Each entry in the matrix is a two-bit number in decimal form. The script computes the bit error rate using `biterr`.

```
x = randint(100,100,4);                    % Original signal

% Create errors to add to ten percent of the elements of x.

% Errors can be either 1, 2, or 3 (not zero).

errorplace = (rand(100,100) > .9);     % Where to put errors

errorvalue = randint(100,100,[1,3]);  % Value of the errors

errors = errorplace.*errorvalue;
```

```
y = rem(x+errors,4);  % Signal with errors added, mod 4

[num_bit,ratio_bit] = biterr(x,y,2)

The output is
num_bit =        1304
ratio_bit =    0.0652
```

Example 3

The commands below convert the parity-check matrix for a Hamming code into the corresponding generator matrix and back again.

```
parmat = hammgen(3)
genmat = gen2par(parmat)
parmat2 = gen2par(genmat) % Ans should be the same as parmat above

The output is

parmat =

     1     0     0     1     0     1     1

     0     1     0     1     1     1     0

     0     0     1     0     1     1     1

genmat =

     1     1     0     1     0     0     0

     0     1     1     0     1     0     0

     1     1     1     0     0     1     0

     1     0     1     0     0     0     1

parmat2 =

     1     0     0     1     0     1     1

     0     1     0     1     1     1     0

     0     0     1     0     1     1     1
```

The above example uses `hammgen` function whose explanation is as follows.

```
h = hammgen(m)
```

The codeword length is `n`. `n` has the form 2^m-1 for some positive integer `m` greater than or equal to 3. The message length, `k`, has the form `n-m`. `hammgen` function produces an `m`-by-`n` parity-check matrix for a Hamming code having codeword length $n = 2^{m-1}$. The input `m` is a positive integer greater than or equal to 3. The message length of the code is `n-m`.

You can also provide your own generator matrix as input.

Example 4

The next example creates a linear code, adds noise, and then decodes the noisy code. It uses the `decode` function.

```
n = 3; k = 2;                       % A (3,2) linear code

msg = randint(100,k,[0,1]);         % 100 messages, k bits each

code = encode(msg,n,k,'linear/binary');% Add noise.

noisycode = rem(code + randerr(100,n,[0 1;.7 .3]), 2);

newmsg = decode(noisycode,n,k,'linear');

% Try to decode.

% Compute error rate for decoding the noisy code.

[number,ratio] = biterr(newmsg,msg);

disp(['The bit error rate is ',num2str(ratio)])

The output is below.
```

Your error rate results might vary because the noise is random.

```
The bit error rate is 0.08
```

EXERCISE:

9.1 Apply table lookup decoding to (7,4) hamming code. The P submatrix is given as

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

a) Write the value of n and k.

b) Create G and H matrices.

c) Tabulate the code words through simulation and write their corresponding weights.

Message	Code Vector	Weights
0000		
0001		
0010		
0011		
0100		
0101		
0110		
0111		
1000		
1001		
1010		
1011		
1100		
1101		
1110		
1111		

d) The minimum hamming distance = _____.

e) The no. of errors this code can detect is

f) The no. of errors this code can correct is

g) Find syndromes for all possible single error patterns.

S	E
0 0 0	**0 0 0 0 0 0 0**
1 0 1	**1 0 0 0 0 0 0**

h) Suppose a received word happens to have two errors, such that E = (1 0 0 0 0 1 0). Calculate the syndrome.

i) What will be the error pattern (write from table)?

j) Write the corrected word.

k) What observations do you make? How many errors will the decoded output word contain?

9.2 Write MATLAB code to construct lookup table for the (6,3) block code generated by the submatrix

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

9.3 Consider a (5,3) block code obtained by deleting the last column of the P submatrix in above question. Construct the lookup table and show that this code could be used for error detection and not correction. Also write MATLAB code to implement the given functionality.

9.4 A (7,4) LBC has generator matrix G given by

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

a) Determine the PCM for above matrix.

b) Write MATLAB code to encode all possible data words and then decode all the encoded words.

c) Insert double error in any code block. Decode manually and through simulation.

d) The bit error rate for single error in first two code blocks is

e) Insert single error in every code block. Write MATLAB code to find bit error rate.

Lab Session 10

Introduction to Cyclic codes

OBJECTIVES

1. To understand the basics of Cyclic codes.
2. Familiarization with Encoding and Decoding techniques for the Cyclic codes.

THEORY

Cyclic codes are a subclass of linear block codes with a cyclic structure that leads to more practical implementation. Block codes used in forward error correction systems are almost always cyclic codes. In cyclic coding, data is sent with a checksum. At the receiver side checksum is recalculated. It should match the one that was sent.

To describe a cyclic code, we express an arbitrary n-bit vector in the form

$$\mathbf{X = (x_{n-1}\, x_{n-2}\, \ldots\ldots\ldots\ldots x_1\, x_0)}$$

Now suppose that X has been loaded into a shift register with feedback connection from the first to last stage. Shifting all bits one position to the left yields the cyclic shift of X, written as

$$\mathbf{X' = (x_{n-2}\, x_{n-3}\, \ldots\ldots\ldots\ldots x_1\, x_0\, x_{n-1})}$$

A second shift produces

$$\mathbf{X'' = (x_{n-3}\, \ldots\ldots\ldots\ldots x_1\, x_0\, x_{n-1}\, x_{n-2})}$$ and so forth.

A linear code is cyclic if every cyclic shift of a code vector X is another vector in the code. Since cyclic codes are linear block codes, all the properties of linear block codes apply to cyclic codes.

Cyclic codes are used in applications where burst errors can occur. Burst error is an error in which a group of adjacent bits is affected.

ENCODING OF CYCLIC CODES

Direct Approach:

Let M(x) is the message polynomial and G(x) is the generator polynomial. Then the code vector V(x) can be encoded as

$$\mathbf{V(x) = M(x).\, G(x)}$$

A linear code is cyclic if every cyclic shift of a code vector X is another vector in the code.

Example 1

The **(6, 2)** repetition code having valid code **C = {[000000], [010101], [101010], [111111]}** is cyclic, since a cyclic shift of any of its code vectors results in a vector that is an element of C.

Example 2

The **(5, 2)** LBC is defined by the generator matrix **G = [1 0 1 1 1; 0 1 1 0 1]** is a single error correcting code. We need to determine if it is cyclic.

The valid code words are

C = {[00000], [01101], [10111], [11010]}

The given code is NOT a cyclic code because cyclic shift of **(11010)** results in **(10101)** which is not a valid code word. Also cyclic shift of **(10111)** results in **(01111)** which is not a member of C.

It is convenient to think of words as polynomials rather than vectors. Since the code is binary, the coefficients will be 0 or 1. The degree of polynomial equals to its highest exponent. For example the degree of the polynomial obtained from (1 0 1 1) is 3

Example 3

Express (1101) as polynomial. We always write most significant digit on the left.

(1101) can be expressed as $\mathbf{1.x^3 + 1.x^2 + 0.x^1 + 1.x^0}$ **or** $\mathbf{x^3 + x^2 + 1}$

Special case: We don't allow bit string = all zeros.

110001 represents:
$\mathbf{1.x^5 + .x^4 + 0.x^3 + 0.x^2 + 0.x^1 + 1.x^0 = x^5 + x^4 + x^0}$

The order of a polynomial is the power of the highest non-zero coefficient. This is polynomial of order 5.

A cyclic code with the generator polynomial of degree `(n-k)` detects all burst errors affecting `n-k` bits or less.

Where

`n` = number of bits in code word.

`k` = number of bits in data word.

Encoding involves multiplying data Polynomial (message polynomial) by generator polynomial.

$$\mathbf{V(x) = M(x).\ G(x)}$$

Calculations are performed as

Multiplication modulo2 = AND

Addition modulo2 = XOR

Example 4

Given (6, 3) code with G(x) = (1 1 0 1). Encode M(x) = (101)

$V(x) = M(x).\ G(x)$

$V(x) = (x^3 + x^2 + 1).\ (x^2 + 1)$

$V(x) = x^5 + x^3 + x^4 + x^2 + x^2 + 1$

$V(x) = 1.x^5 + 1.x^4 + 1.x^3 + (1 \oplus 1)\ x^2 + 0.x + 1$

$V(x) = 1.x^5 + 1.x^4 + 1.x^3 + 0.x^2 + 0.x + 1$

V(x) = (1 1 1 0 0 1)

The degree of polynomial is defined to be the highest power of `x` in the polynomial for which the coefficient of `x` is not zero. The degree of polynomial `M(x)` is denoted by `deg [M(x)]`. If `M` is our set of `k` bit source vectors, the highest degree polynomial in the set of polynomials `M(x)` that describes our source vectors is `k-1`. If `deg [G(x)] = r`, then the highest degree of our set of code polynomials `V(x)` is `n-1` or `k-1+r`. Therefore `deg [G(x)]` tells us the number of check bits in the code.

Property of generator Polynomial

`G(x)` is the generator polynomial for a linear cyclic code of length `n` if and only if `G(x)` divides $1+x^n$ without a remainder.

Example 5

Given (7, 4) cyclic code. Find a generator polynomial for a code of length `n` = 7 for encoding of data of length `k` = 4.

Solution:

`G(x)` should be a degree 7 – 4 = 3 and should divide $1 + x^n$ without a remainder.

$1 + x^7$ can be factorized as

$1 + x^7 = (1+x+x^3)\ (1+x^2+x^3)\ (1+x)$

So we can choose for `G(x)` either $(1+x+x^3)$ or $(1+x^2+x^3)$.

Systematic form:

Let
$U(x) = u_{k-1}\ x^{k-1} + u_{k-2}\ x^{k-2} + ... + u_1\ x + u_0$ be an information polynomial and
$G(x) = g_{n-k}\ x^{n-k} + g_{n-k-1} x^{n-k-1} + ... + g_1\ x + g_0$ be a generator polynomial of an (n, k) cyclic code. The code polynomial is given by
$V(x) = x^{n-k}\ U(x) / G(x)$
$= V_s(x) / G(x)$

Where
$V_s(x) = x^{n-k} U(x)$

Systematic encoding of an (n, k) cyclic code consists of multiplying U(x) by x^{n-k}, then computing the remainder b(x) of x^{n-k} U(x) / G(x), and forming the code polynomial V(x). A linear (n – k)-stage shift register with feedback connections circuit can simultaneously accomplish the multiplication and division tasks.

This systematic encoding form a class of cyclic codes called "Cyclic Redundancy Check".

Cyclic Redundancy check or CRC codes are error detecting codes typically used in automatic repeat request systems. CRC codes have no error correction capability but they can be used in combination with an error correcting code. When used in this fashion, the error control system is in the form of a *concatenated* code. CRC codes are generated and their syndromes are calculated in precisely the same manner as other cyclic codes. Since these codes are error detecting rather than error correcting, there is no error correction circuit. Instead, the syndrome of the received block is checked to see if it is ZERO (no error detected) or NON ZERO (error detected).

Example 6

Consider a message 110010 represented by the polynomial
$M(x) = x^5 + x^4 + x$

Consider a *generating polynomial*

$G(x) = x^3 + x^2 + 1$ (1101)

This is used to generate a 3 bit CRC = C(x) to be appended to M(x).

STEPS:

1. Multiply M(x) by x^3 (highest power in G(x)) i.e. Add 3 zeros. **110010000**
2. Divide the result by G(x). The remainder = C(x). 1101 long division into 110010000 (with subtraction mod 2) = **100100** remainder **100**

 Special case: This won't work if bit string = all zeros. We don't allow such an M(x).

 Transmit **110010000 + 100**. To be precise, transmit: **$T(x) = x^3M(x) + C(x)$** = **110010100**
3. Receiver end: Receive T(x). Divide by G(x), should have remainder 0.

Example 7

The following example shows the long division of message word (1101011011) by the generator (10011)

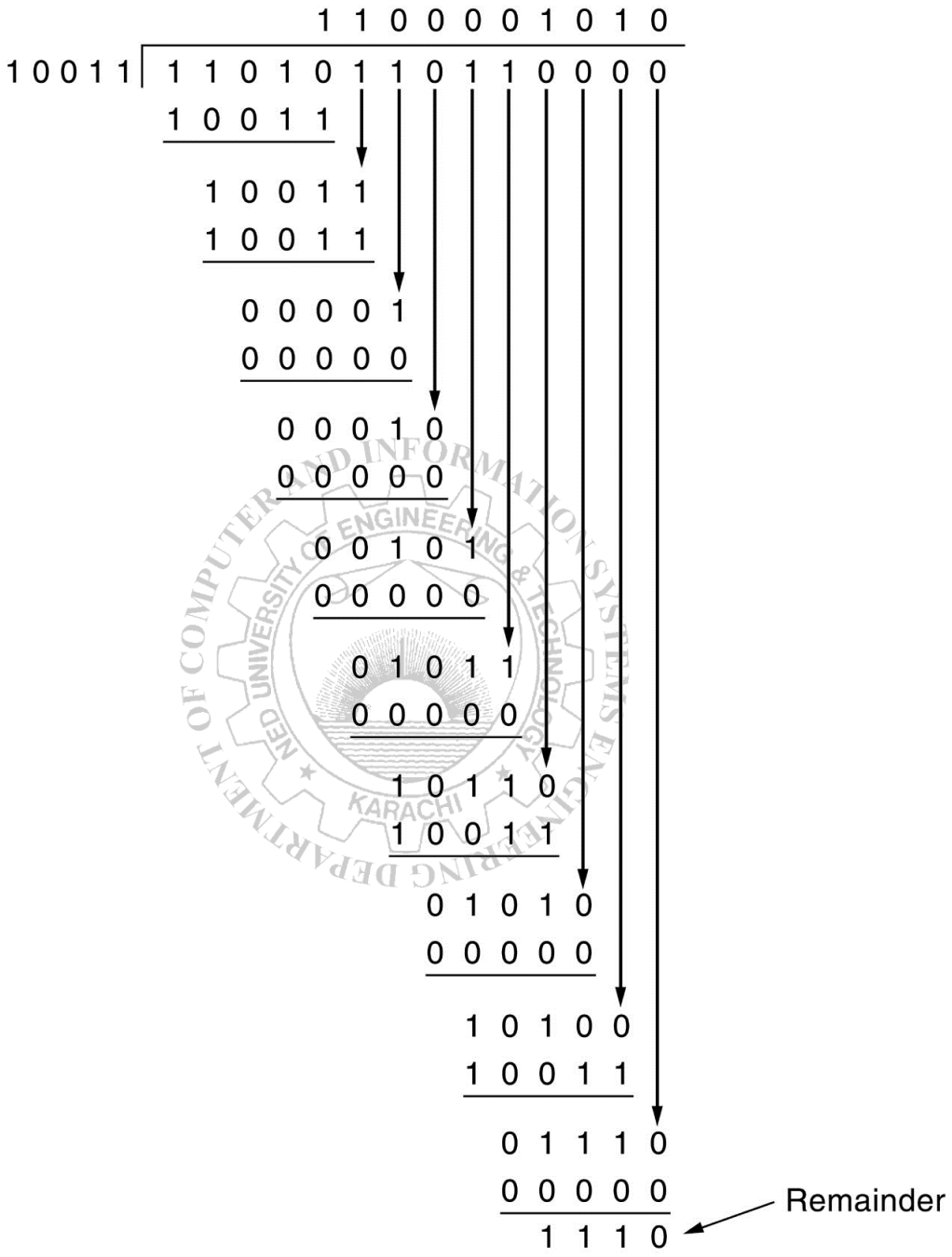
The remainder obtained is appended (added) to the frame to form the transmitted vector.

Highest power of generator polynomial is '4'

Frame : 1 1 0 1 0 1 1 0 1 1

Generator: 1 0 0 1 1

Message after 4 zero bits are appended: 1 1 0 1 0 1 1 0 1 1 0 0



Transmitted frame: 1 1 0 1 0 1 1 0 1 1 1 1 1 0

Receiver end: Receive T(x). Divide by G(x), should have remainder 0; verify it yourself.

DECODING

Let T(x) and E(x) be the code polynomial of a systematic cyclic code and an error polynomial of degree `n - 1` or less, respectively. An error is the same as adding some E(x) to T(x) for example adding 1010011000001110000 will flip the bits at the locations where "1" is in the error bit string. Instead of T(x) arriving, T(x) $\oplus$ E(x) arrives.

So R(x) = T(x) $\oplus$ E(x)

In general, each 1 bit in E(x) corresponds to a bit that has been flipped in the message. If there are `k` 1 bits in E(x), `k` single-bit errors have occurred. A burst error looks like 1....1

Far end receives T(x) + E(x) = R(x). T(x) is multiple of G(x) (remainder zero). Hence remainder of (T(x) $\oplus$ E(x)) by G(x) = remainder of E(x) by G(x). For example remainder when divide (1000 + n) by 10 = remainder when you divide n by 10

Here the remainder of E(x) / G(x) is called syndrome vector denoted by S(x). If remainder when you divide E(x) by G(x) is zero, the error will *not* be detected.
In general, if E(x) is a multiple of G(x), the error will not be detected. Otherwise, it will. *All* other error patterns will be caught.

MATLAB FUNCTIONS

1) Encode
Function: Block encoder

Syntax
```
Code = encode (msg,n,k,'cyclic/fmt',genpoly)
```

Description

`Encode` function encodes `msg` and creates a systematic cyclic code. `genpoly` is a row vector that gives the coefficients, in order of ascending powers, of the binary generator polynomial. The default value of `genpoly` is `cyclpoly (n,k)`. This function is explained below. By definition, the generator polynomial for an [n, k] cyclic code must have degree n-k and must divide x^n-1. You are already familiar with the input arguments `'msg', 'n'` and `'k'`.

2) Cyclpoly
Function: Produce generator polynomials for a cyclic code.

Syntax
```
pol = cyclpoly (n, k)
```

Description

In MATLAB a polynomial is represented as a row containing the coefficients in order of ascending powers. The results you would get will be in ascending order of coefficients while in manual calculation the results will be in descending order. It does not matter what convention you follow but it should be consistent. `Cyclpoly` function returns the row vector representing one nontrivial generator polynomial for a cyclic code having codeword length `n` and message length `k`.

3) Decode
Function: Block decoder

Syntax

```
msg = decode (code, n ,k, 'cyclic/fmt', genpoly, trt)
[msg, err] = decode (...)
[msg, err, ccode] = decode (...)
[msg, err, ccode, cerr] = decode (...)
```

Description

`decode` function decodes the cyclic code '`code`' and tries to correct errors using the decoding table '`trt`', where `trt` is a 2^(n-k)-by-n matrix. '`genpoly`' is a row vector that gives the coefficients, in order of ascending powers, of the binary generator polynomial of the code. The default value of `genpoly` is `cyclpoly (n, k)`. By definition, the generator polynomial for an [n, k] cyclic code must have degree `n-k` and must divide x^n-1.

'`trt`' uses `syndtable` to create the syndrome decoding table associated with the method's parity-check matrix. `Syndtable` function is explained in lab session 4. The return parameters '`err`', '`ccode`' and '`cerr`' have the same purpose as explained in lab session 4.

Example 8
The following example creates a cyclic code, adds noise, and then decodes the noisy code. It uses the decode function.

```
n = 3; k = 2; % A (3, 2) cyclic code
msg = randint(100,k,[0,1]); % 100 messages, k bits each
Code = encode (msg, n, k, 'cyclic/binary'); % Add noise.
noisycode = rem(code + randerr (100,n,[0 1;.7 .3]), 2);
newmsg = decode(noisycode, n, k, 'cyclic'); % Try to decode.
% Compute error rate for decoding the noisy code.
[Number, ratio] = biterr (newmsg, msg);
disp (['The bit error rate is ',num2str(ratio)])
```

Example 9

The example below illustrates the use of 'err' and 'cerr'. The script encodes a five-bit message using a cyclic code. The codeword has fifteen bits. No Error is added to the codeword. Then decode is used to recover the original message.

```
m = 4
n = 2^m-1              % Codeword length is 15.
k = 5                  % Message length
msg = ones(1,5)        % One message of five bits.

Code = encode (msg,n,k,'cyclic')  % Encode the message.

noisycode = code           % No error

% decodes and tries to correct the errors.
[newmsg,err,ccode,cerr] = decode (noisycode,n,k,'cyclic')
disp ('Transpose of err is')
disp (err')
disp ('Transpose of cerr is')
disp (cerr')
```

The Galois Field Computation in MATLAB

A Galois field is an algebraic field that has a finite number of members. Galois fields having `2^m` members are used in error-control coding and are denoted `GF (2^m)` where `m` is an integer between `1` and `16`. This portion describes how to work with fields that have `2^m` members with `m = 0`.

4) gf

Function: Creates a Galois field array.

Syntax

```
x_gf = gf(x,m)
x_gf = gf(x)
```

Description

`x_gf = gf(x,m)` creates a Galois field array from the matrix `x`. The Galois field has `2^m` elements, where `m` is an integer between 1 and 16. The elements of `x` must be integers between `0` and `2^m-1`. The output `x_gf` is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. As a result, when you manipulate `x_gf` using operators or functions such as + or det, MATLAB works within the Galois field you have specified.

`X_gf = gf(x)` creates a `GF (2)` array from the matrix `x`. Each element of `x` must be `0` or `1`.

Example 10:

Let
```
g = [0 1 1; 1 0 1; 1 1 0]
d = [0 1 0]
```
Run the following statement
```
c = d*g
```
```
Output:
c =     1     0     1
```

Using `d = [1 0 1]` rerun the above statement.

```
Output:
c =     1     2     1
```

When you perform modulo 2 multiplications on two binary vectors, you get a binary vector as result. This is not the case here because MATLAB considers these vector elements as integers. For this purpose we convert the vectors into Galois field of order 2.

Now convert all the given vectors and g matrix to Galois field and do the computation again.

```
G=gf (g)
D=gf (d)
c=D*G
```

Array elements =

1 0 1

5) conv
Function: Convolution and polynomial multiplication

Syntax:
```
w = conv (u,v)
```

Description:
`conv` function convolves vectors `u` and `v`. Algebraically, convolution is the same operation as multiplying the polynomials whose coefficients are the elements of `u` and `v`.

Example 11:

```
m = [1 1 0 0 1 0]
g = [1 1 0 1]
```
Compute c = m*g manually.
Answer: 101101010.
Run the following statement.
```
c = m*g
```

Note the error you get.
Convert `m` and `g` in to Galois field arrays.

```
M = gf(m)
G = gf(g)
```

Rerun the statement `C=M*G`

Now note the error that you get. Despite the fact that we have converted `m` and `g` into Galois field, still multiplication is not possible because sizes of `m` and `g` don't match.
Now use `conv` function

```
C = conv (M, G)
```

Note that now you get the correct result.

6) deconv
Function: Deconvolution and polynomial division.

Syntax:

```
[q,r] = deconv (v,u)
```

Description:
`Deconv` function deconvolves vector `u` out of vector `v`, using long division.

The quotient is returned in vector `q` and the remainder in vector `r` such that
`v = conv (u,q)+r.`

If `u` and `v` are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials, and deconvolution is polynomial division.
The result of dividing `v` by `u` is quotient `q` and remainder `r`.

Example 12:

```
m=[1 1 0 0 1 0]
g=[1 1 0 1]
```

Using long division we compute `(x^3 * m) / g` manually.

Answer:

```
Quotient: 100100
Remainder:100
```

Now perform the same functionality using MATLAB.

```
M=gf(m)
G=gf(g)
h=[1 0 0 0]            % or x^3 that is highest power of g(x)
H=gf(h)
C=conv(H,M)            % multiplying x3 * m(x)
[q,r]=deconv(C,G)      % dividing [x3*m(x)] by g(x)
```

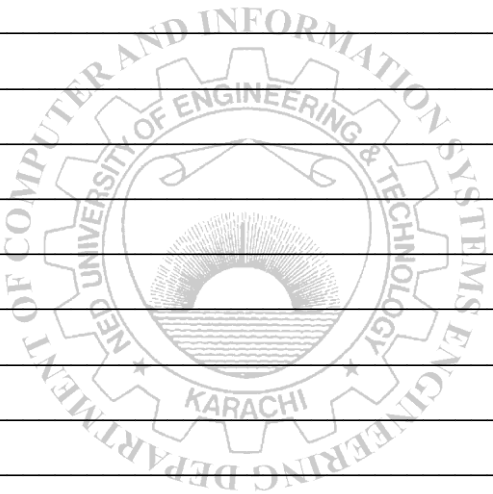
See that the results you get in both the cases match.

EXERCISES

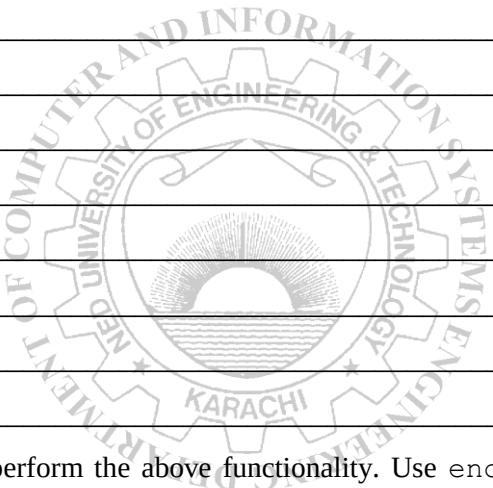
10.1 The generator polynomial of a (7, 4) cyclic code is given by

$$G(x) = 1 + x + x^3$$

Find V by direct method as well as the systematic approach if `M(x) = (1 0 1 0)`.
Write code to implement the above functionality in MATLAB using `encode` function and then `conv` function.



10.2 Consider the cyclic (7, 4) cyclic code generated by `G(x) = x^3 + x + 1`.
a) Use long division to calculate the check bit polynomial `C(x)` when `M = (1 1 0 0)`. Also write the complete code polynomial.

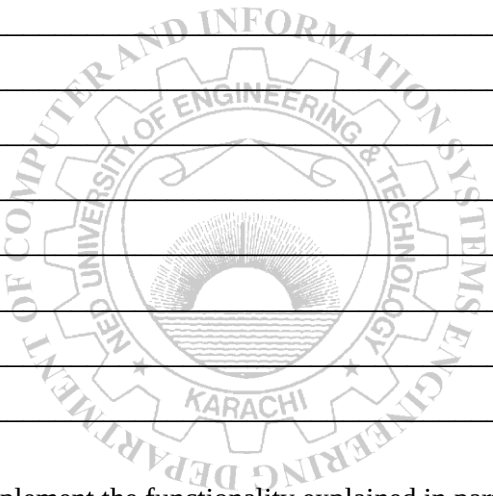


b) Write MATLAB code to perform the above functionality. Use `encode/decode` functions and then `conv/deconv` functions.

10.3 The generator polynomial of (7, 4) cyclic codes is given by $G(x) = 1 + x^2 + x^3$. Find V by the systematic approach if M(x) = (1 0 0 1).Let the received word is [1 1 1 0 0 1 1]. Find the syndrome vector.

10.4 Consider the cyclic (7, 4) hamming code generated by $G(x) = x^3 + x + 1$.

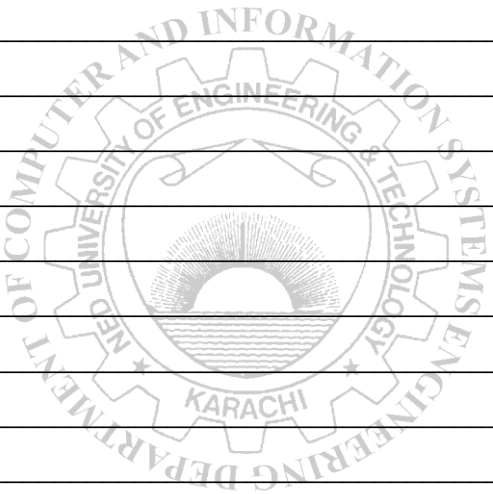
a) Use systematic approach to calculate the check bit polynomial C(x) when M = (1 1 0 1). Also write the complete code polynomial.



b) Write MATLAB code to implement the functionality explained in part (a).

c) Find the syndrome vectors for the following erroneous received words.

i. 1010101
ii. 1000001
iii. 1011001
iv. 1010000
v. 1010011

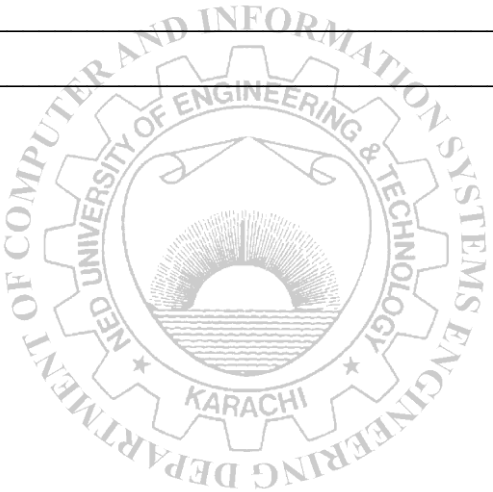


10.5 Consider (6, 3) cyclic code. Create MATLAB code to perform the following tasks.
a) Create generator polynomial.

b) Write all possible code words and encode using '`encode`' function.

c) Use '`decode`' function to decode all the encoded words.

d) Induce single error in all the encoded words and then decode.
Record the output that you get.



Lab Session 11

Familiarization with Huffman Coding Techniques

OBJECTIVE

To understand Huffman Coding Techniques and its Application.

THEORY

Huffman coding

Huffman coding is a compression method that converts characters into variable length bit strings. Most-frequently occurring characters are converted to shortest bit strings; least frequent, the longest.

In computer science and information theory, **Huffman coding** is an encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol. It was developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes."

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix-free code (sometimes called "prefix codes") (that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol) that expresses the most common characters using shorter strings of bits than are used for less common source symbols. Huffman was able to design the most efficient compression method of this type: no other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code.

Basic technique

Assume you have a source generating 4 different symbols {a_1, a_2, a_3, a_4} with probability {0.4; 0.35; 0.2; 0.05}. Generate a binary tree from left to right taking the two less probable symbols, putting them together to form another equivalent symbol having a probability that equals the sum of the two symbols. Keep on doing it until you have just one symbol. Then read the tree backwards, from right to left, assigning different bits to different branches as shown in fig 11.1.

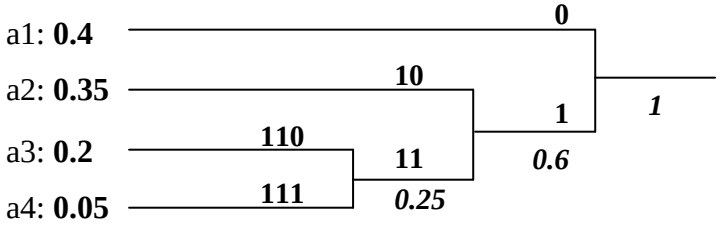


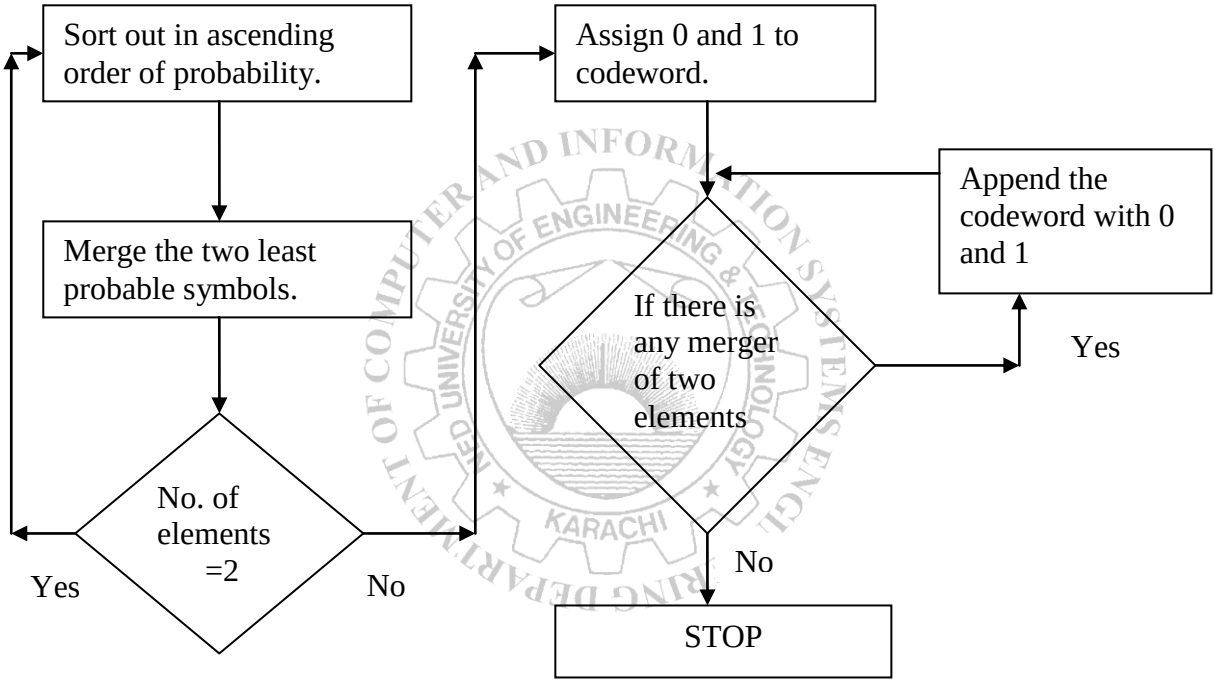
Figure 11.1: Huffman Tree

The final Huffman code is:

Symbol	Code
a1	0
a2	10
a3	110
a4	111

The standard way to represent a signal made of 4 symbols is by using 2 bits/symbol, but the entropy of the source is 1.73 bits/symbol. If this Huffman code is used to represent the signal, then the entropy is lowered to 1.83 bits/symbol.

Flow Chart of Huffman Algorithm:



a) The Prefix Property

Data encoded using Huffman coding is uniquely decodable. This is because Huffman codes satisfy an important property called the prefix property. This property guarantees that no code word is a prefix of another Huffman code word. For example, 10 and 101 cannot simultaneously be valid Huffman code words because the first is a prefix of the second. Thus, any bit stream is uniquely decodable with a given Huffman code. We can see by inspection that the code words we generated are valid Huffman code words. To see why the prefix property is essential, consider the code words given below in which "e" is encoded with 110 which is a prefix of "f".

character	a	b	c	d	e	f
code word	0	101	100	111	110	1100

The decoding of 11000100110 is ambiguous:

11000100110 ➔ face

11000100110 ➔ eaace

i) Encode (compress) the message **tenseas** using the following code words:

Character	a	e	l	n	o	s	t
Huffman codeword	110	10	0110	111	0111	010	00

Answer: Replace each character with its codeword:
001011101010110010

ii) Decode (decompress) each of the following encoded messages, if possible, using the Huffman codeword tree obtained above
 a) 0110011101000
 b) 11101110101011

Answer: Decode a bit-stream by starting at the root and proceeding down the tree according to the bits in the message (0 = left, 1 = right). When a leaf is encountered, output the character at that leaf and restart at the root. If a leaf cannot be reached, the bit-stream cannot be decoded.

(a) 0110**0111**010**00**
0110 corresponds to **l**, **0111** corresponds to **o**, **010** is the code for **s** and **00** is the code for **t**.
So the given stream is decoded to the word **lost**.

(b) 111**0111**010**10**11
Here **111** correspond to **n**, **0111** corresponds to **o**, **010** is the code for **s**.
The decoding fails because the corresponding node for 11 is not a leaf.

Assume that the numbers of character-codeword pairs are included at the beginning of the binary file containing the compressed message in the format as shown in figure 11.4.

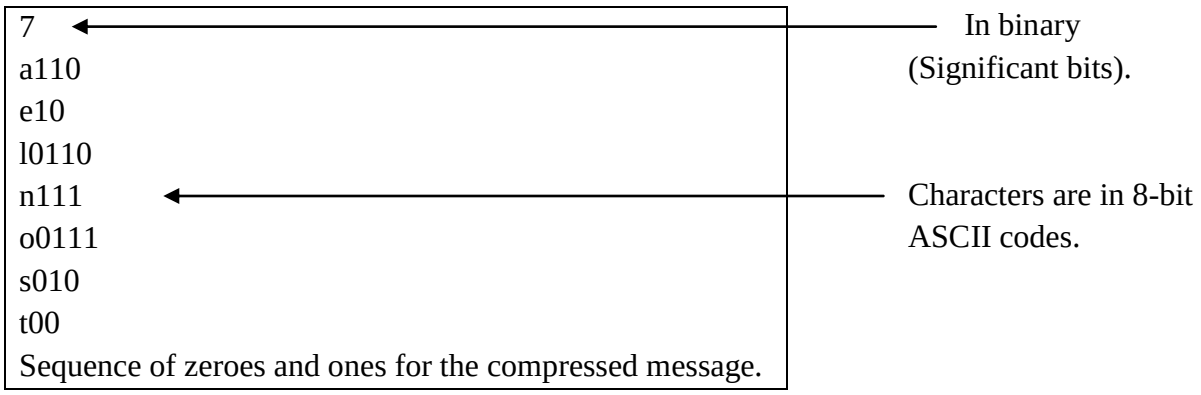


Figure 11.4: character-code word pairs at the beginning of the binary file containing the compressed message

Number of bits for the transmitted file = bits (7) + bits (characters) + bits (code words) + bits (compressed message)

= 3 + (7*8) +21 + 696
= 776

Compression Ratio = bits for ASCII representation / number of bits transmitted
= 2088 / 776
= 2.69

Thus, the size of the transmitted file is 100 / 2.69 = 37.17 % of the original ASCII file (i.e. 37 % compression has been achieved)

Huffman Coding in MATLAB

In MATLAB Huffman code dictionary associates each data symbol with a codeword. It has the property that no codeword in the dictionary is a prefix of any other codeword in the dictionary. The `huffmandict`, `huffmanenco`, and `huffmandeco` functions support Huffman coding and decoding.

Creating a Huffman Code Dictionary

```
dict = huffmandict(symbols, p)
```

Huffman coding requires statistical information about the source of the data being encoded. In particular, the `p` input argument in the `huffmandict` function lists the probability with which the source produces each symbol in its alphabet.

For example, consider a data source that produces 1s with probability 0.1, 2s with probability 0.1, and 3s with probability 0.8. The main computational step in encoding data from this source using a Huffman code is to create a dictionary that associates each data symbol with a codeword. The commands below create such a dictionary and then show the codeword vector associated with a particular value from the data source.

```
symbols = [1 2 3]; % Data symbols
p = [0.1 0.1 0.8]; % Probability of each data symbol
dict = huffmandict(symbols, p) % Create the dictionary.
dict {1,:} % Show first row of the dictionary.
dict {2,:} % Show second row of the dictionary.
dict {3,:} % Show third row of the dictionary.
```

The output below shows that the most probable data symbol, 3, is associated with a one-digit codeword, while less probable data symbols are associated with two-digit code words. The output also shows, for example, that a Huffman encoder receiving the data symbol 1 should substitute the sequence 11.

```
dict =
    [1]    [1x2 double]
    [2]    [1x2 double]
    [3]    [          0]

ans =
     1

ans =
     1     1

ans =
     2

ans =
     1     0

ans =
     3

ans =
     0
```

Creating and Decoding a Huffman Code

The example below performs Huffman encoding and decoding using `huffmanenco` and `huffmandeco` functions. A source is used whose alphabet has three symbols. Notice that the `huffmanenco` and `huffmandeco` functions use the dictionary that `huffmandict` created.

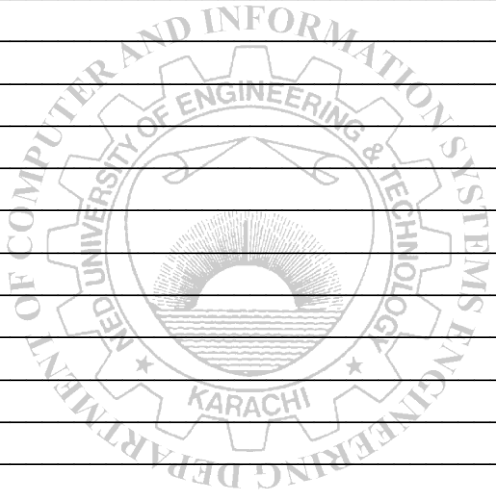
```
sig = repmat([3 3 1 3 3 3 3 3 2 3],1,50); % Data to encode
symbols = [1 2 3]; % Distinct data symbols appearing in sig
p = [0.1 0.1 0.8]; % Probability of each data symbol
dict = huffmandict(symbols,p); % Create the dictionary.
hcode = huffmanenco(sig,dict); % Encode the data.
dhsig = huffmandeco(hcode,dict); % Decode the code.
```

EXERCISES:

11.1 Five source symbols of the alphabet of a source and their probabilities are shown

Symbol	Probability
S0	**0.4**
S1	**0.2**
S2	**0.2**
S3	**0.1**
S4	**0.1**

a) Build the Huffman code tree for the message and find the codeword for each character.



b) Write MATLAB code to obtain Huffman coding for the given characters.

11.2 Construct a Huffman code for the source X emitting symbols with probabilities as shown below.

Symbol	Probability	Symbol	Probability
A	**0.4**	**B**	**0.3**
C	**0.1**	**D**	**0.1**
E	**0.06**	**F**	**0.04**

Determine:

a) The code word for each symbol.

b) The average number of transmitted binary digits per code word.

c) Write MATLAB code to obtain Huffman code for the given source.

Section Four

Signal Processing

Lab Session 12

Audio Signal Processing

OBJECTIVE

To get familiarize with various processing techniques on audio signals.

THEORY

Audio processing covers many diverse fields, all involved in presenting sound to human listeners. Three areas are prominent: (1) high fidelity music reproduction, such as in audio compact discs, (2) voice telecommunications, another name for telephone networks, and (3) synthetic speech, where computers generate and recognize human voice patterns. While these applications have different goals and problems, they are linked by a common umpire: the human ear. Digital Signal Processing has produced revolutionary changes in these and other areas of audio processing.

The perception of a continuous sound, such as a note from a musical instrument, is often divided into three parts: loudness, pitch, and timbre (pronounced "timber"). Loudness is a measure of sound wave intensity, as previously described. Pitch is the frequency of the fundamental component in the sound, that is, the frequency with which the waveform repeats itself. While there are subtle effects in both these perceptions, they are a straightforward match with easily characterized physical quantities.

Sound Quality vs. Data Rate:

When designing a digital audio system there are two questions that need to be asked: (1) how good does it need to sound? And (2) what data rate can be tolerated? The answer to these questions usually results in one of the three categories. First, high fidelity music, where sound quality is of the greatest importance and almost any data rate will be acceptable. Second, telephone communication, requiring natural sounding speech *and* a low data rate to reduce the system cost. Third, compressed speech, where reducing the data rate is very important and some unnaturalness in the sound quality can be tolerated. This includes military communication, cellular telephones, and digitally stored speech for voice mail and multimedia.

High fidelity music systems sample fast enough (44.1 kHz), and with enough precision (16 bits), that they can capture virtually all of the sounds that humans are capable of hearing. This magnificent sound quality comes at the price of a high data rate, 44.1 kHz × 16 bits = 706k bits/sec. This is pure brute force.

Whereas music requires a bandwidth of 20 kHz, natural sounding speech only requires about 3.2 kHz. Even though the frequency range has been reduced to only 16% (3.2 kHz out of 20 kHz), the signal still contains 80% of the original sound information (8 out of 10 octaves). Telecommunication systems typically operate with a sampling rate of about 8 kHz, allowing natural sounding speech, but greatly reduced music quality. You are probably already familiar with this difference in sound quality: FM radio stations broadcast with a bandwidth of almost 20 kHz, while AM radio stations are limited to about 3.2 kHz. Voices sound normal on the AM stations, but the music is weak and unsatisfying.

Voice-only systems also reduce the precision from 16 bits to 12 bits per sample, with little noticeable change in the sound quality. This can be reduced to only 8 bits per sample if the quantization step size is made unequal. This is a widespread procedure called **companding.**

Speech Synthesis and Recognition:

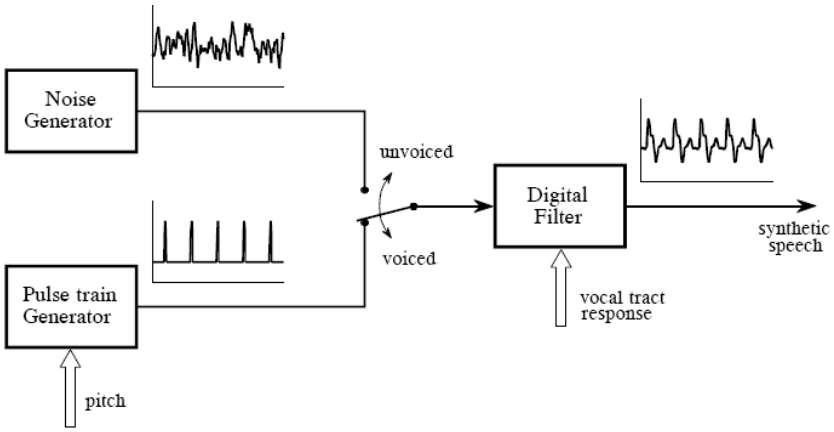


Figure 12.1: Voice Synthesizer

For detail about its working, see chapter 22 from the book: "The Scientist and Engineer's Guide to Digital Signal Processing" by Steven W. Smith, Ph.D.

MATLAB Instructions:

1. To generate a single tone (a single sinusoidal signal with frequency F):
As we change the frequency, the tone or sound varies. Further, to hear the sound, we use the command `sound(signal,Fs)` to play the signal at a sampling rate of Fs. The following code is going to play a single tone:

```
F1=400; T1=1/F1; Fs=40000; Ts=1/Fs; f1=F1/Fs;
t=[0:Ts:50*T1]; A=1;
x1=A*sin(2*pi*F1*t);
sound(x1,Fs)
```

Try the following modifications: vary the amplitude, vary the playing duration period, vary the frequency and vary the sampling frequency. If Fs in not provided in the sound command, it is going to play a default Fs = 8192Hz.

2. To generate multiple tones played in series, write the following code:

```
F1=400; T1=1/F1; Fs=40000; Ts=1/Fs; f1=F1/Fs;
F2=100; F3=200; F4=600; F5=300; F6=800;
t=[0:Ts:0.1]; A=1;
x1=A*sin(2*pi*F1*t); x2=A*sin(2*pi*F2*t); x3=A*sin(2*pi*F3*t);
x4=A*sin(2*pi*F4*t); x5=A*sin(2*pi*F5*t); x6=A*sin(2*pi*F6*t);
y=x1+x2+x3+x4+x5+x6;
z=[x2 x6 x5 x2 x6 x4 x2 x5 x4 x2 x5 x5 x3 x6 x4 x2 x2 x3 x2 x1 x5];
sound(z,Fs)
```

3. Timbre:

Figure 12.2 illustrates two waveforms, each formed by adding a 1 kHz sine wave with amplitude of 1, to a 3 kHz sine wave with amplitude of 0.5. The difference between the two waveforms is that the one shown in (b) has the higher frequency *inverted* before the addition. In spite of the very different time domain waveforms, these two signals sound *identical*. This is because hearing is based on the *amplitude* of the frequencies, and is very insensitive to their *phase*. The *shape* of the time domain waveform is only indirectly related to hearing, and usually not considered in audio systems. **(Try this out)**

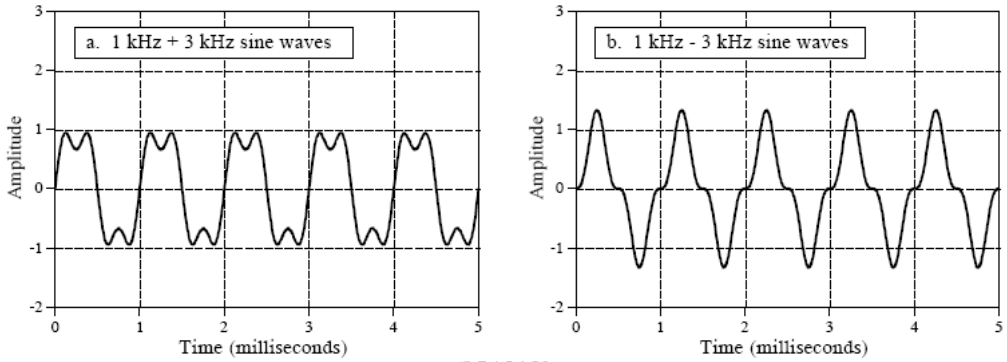


Figure 12.2: Timbre effect

4. Adding noise to a signal:

The following code adds noise (a beep) to a recorded audio signal.

```
load mtlb
sound(mtlb,Fs)
noise = cos(2*pi*3*Fs/8*(0:length(mtlb)-1)/Fs)';
pause
sound(noise,Fs)
u=mtlb+u;
sound(u,Fs)
plot(mtlb)
figure, plot(noise)
figure, plot(u)

mtlb_frequency = abs(fft(mtlb));
noise_frequency = abs(fft(noise));
u_frequency = abs(fft(u));
figure
plot(mtlb_frequency)
hold on
plot(noise_frequency,'r')
plot(u_frequency,'g')
```

EXERCISE:

12.1. Try to simulate violin or piano keys by playing different notes or key strokes respectively. Explain your result below.

12.2. Add more severe noise to your voice signal and see what happens to the signal in both time and frequency domain. Write down your observations.

12.3. Read a wave file and play it in reverse. Further, try to slow it down and play it fast. Write your observations

12.4. Modulate your digital audio signal using QPSK then pass it through AWGN channel and recover your audio signal back.

12.5. Implement the voice synthesizer using SIMULINK (Optional task).

APPENDIX A

An introduction to MATLAB

The purpose of this Appendix is to review MATLAB for those that have used it before, and to provide a brief introduction to MATLAB for those that have not used it before. This is a "hands-on" tutorial introduction. After using this tutorial, you should be able to:

- ✓ Enter matrices
- ✓ Perform matrix operations
- ✓ Make plots
- ✓ Use MATLAB functions
- ✓ Write simple m-files

BACKGROUND:

MATLAB is an interactive program for numerical computation and data visualization. It was originally developed in FORTRAN as a MATrix LABoratory for solving numerical linear algebra problems. The original application may seem boring (except to linear algebra enthusiasts), but MATLAB has advanced to solve nonlinear problems and provide detailed graphics. It is easy to use, yet very powerful. A few short commands can accomplish the same results that required a major programming effort only a few years ago.

MATLAB features a family of add-on application-specific solutions called toolboxes. These toolboxes allow you to learn and apply specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include communications, signal processing, control systems, neural networks, fuzzy logic, simulation, and many others.

Starting MATLAB

On Windows platforms, start MATLAB by double-clicking the MATLAB shortcut icon on your Windows desktop.

MATLAB Documentation

MATLAB provides extensive documentation, in both printed and online format, to help you learn about and use all of its features. If you are a new user, start with Getting Started book. It covers all the primary MATLAB features at a high level, including many examples. The MATLAB online help provides task-oriented and reference information about MATLAB features. MATLAB documentation is also available in printed form and in PDF format.

THE MATLAB ENVIRONMENT

The MATLAB environment consists of five main parts:

1) Development Environment

This is the set of tools and facilities that help you use MATLAB functions and files. Many of these tools are graphical user interfaces. It includes the MATLAB desktop and Command Window, a command history, an editor and debugger, and browsers for viewing help, the workspace, files, and the search path.

SUMMARY OF DESKTOP TOOLS

- **Array Editor:** View array contents in a table format and edit the values.
- **Command Window:** Run MATLAB functions. The main window in which commands are keyed in after the command prompt >>
 Results of most printing commands are displayed in this window.
- **Command History:** View a log of the functions you entered in the Command Window, copy them, execute them, and more. This window records all of the executed commands as well as the date and time when these commands were executed.

 This feature comes very handy when recalling previously executed commands.
 – Previously entered commands can also be re-invoked using **up arrow key** ↑
- **Current Directory Browser:** View files, perform file operations such as open, find files and file content, and manage and tune your files.
- **Help Browser:** View and search the documentation for all your Math Works products.
- **Start Button:** Run tools and access documentation for all of your Math Works products, and create and use MATLAB shortcuts.
- **Workspace Browser:** This window is used to organize the loaded variables and displays the information such as size and class of these variables.
 – View and make changes to the contents of the workspace.

Quitting MATLAB

To end your MATLAB session, select File -> Exit MATLAB in the desktop, or type quit or exit in the Command Window.

2) The MATLAB Mathematical Function Library

This is a vast collection of computational algorithms ranging from elementary functions, like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix Eigen values, Bessel functions and fast Fourier transforms.

3) The MATLAB Language

This is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both rapid creations of small programs as well as large and complex application programs.

4) Graphics

MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. It includes high-level functions for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level functions that allow you to fully customize the appearance of graphics as well as to build complete graphical user interfaces on your MATLAB applications.

5) The MATLAB Application Program Interface (API)

This is a library that allows you to write C and FORTRAN programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files.

GETTING STARTED

This session provides a brief overview of essential MATLAB commands. *You will learn this material more quickly if you use MATLAB interactively as you are reviewing this manual*. The MATLAB commands will be shown in the following font style:

`Monaco font`
the prompt for a user input is shown by the double arrow
»

MATLAB has an extensive on-line help facility.
For example, type `help pi` at the prompt
```
» help pi
PI PI = 4*atan(1) = 3.1415926535897...
```
so we see that MATLAB has the number π "built-in".

As another example
```
» help exp
EXP EXP(X) is the exponential of the elements of X, e to the X.
```

Sometimes you do not know the exact command to perform a particular operation. In this case, one can simply type
```
» help
```

and MATLAB will provide a list of commands (and m-files, to be discussed later) that are available. If you do not know the exact command for the function that you are after, another useful command is `lookfor`. This command works somewhat like an index. If you did not know the command for the exponential function was `exp`, you could type
```
» lookfor exponential
EXP Exponential.
EXPM Matrix exponential.
EXPM1 Matrix exponential via Pade' approximation.
EXPM2 Matrix exponential via Taylor series approximation.
```

```
EXPM3 Matrix exponential via eigenvalues and eigenvectors.
EXPME Used by LINSIM to calculate matrix exponentials.
```

a) Entering Matrices

The best way for you to get started with MATLAB is to learn how to handle matrices. Start MATLAB and follow along with each example. You can enter matrices into MATLAB by entering an explicit list of elements or generating matrices using built-in functions.

You only have to follow a few basic conventions: Separate the elements of a row with blanks or commas. Use a semicolon "`;`" to indicate the end of each row. Surround the entire list of elements with square brackets "`[ ]`".
Consider the following vector, `x` (recall that a vector is simply a matrix with only one row or column)

```
» x = [1,3,5,7,9,11]

x = 1 3 5 7 9 11
```

Notice that a row vector is the default. We could have used spaces as the delimiter between columns

```
» x = [1 3 5 7 9 11]
x = 1 3 5 7 9 11
```

There is a faster way to enter matrices or vectors that have a linear pattern. For example, the following command creates the previous vector

```
» x = 1:2:11
```

(here what does '2' indicate? Will be discussed in proceeding lab sessions)

```
x = 1 3 5 7 9 11
```

Transposing a row vector yields a column vector (`'` is the transpose command in MATLAB)

```
» y = x'
y =   1
      3
      5
      7
      9
      11
```

Say that we want to create a vector `z`, which has elements from 5 to 30, by 5's

```
» z = 5:5:30
z = 5 10 15 20 25 30
```

If we wish to suppress printing, we can add a semicolon (`;`) after any MATLAB command

```
» z = 5:5:30;
```

The `z` vector is generated, but not printed in the command window. We can find the value of the third element in the `z` vector, `z(3)`, by typing

```
» z(3)
ans = 15
```

(Notice that a new variable, `ans`, was defined automatically.)

The MATLAB Workspace

We can view the variables currently in the workspace by typing

```
» who
Your variables are:
ans     x    y      z

leaving 621420 bytes of memory free.
```

More detail about the size of the matrices can be obtained by typing

```
» whos
```

Name	Size	Total	Complex
ans	1 by 1	1	No
x	1 by 6	6	No
y	6 by 1	6	No
z	1 by 6	6	No

```
Grand total is (19 * 8) = 152 bytes,
leaving 622256 bytes of memory free.
```

We can also find the size of a matrix or vector by typing

```
» [m,n]=size(x)
m =1
n =6
```

where m represents the number of rows and n represents the number of columns.
If we do not put place arguments for the rows and columns, we find

```
» size(x)
ans = 1  6
```

Since `x` is a vector, we can also use the length command

```
» length(x)
ans = 6
```

It should be noted that MATLAB is case sensitive with respect to variable names. An `X` matrix can coexist with an `x` matrix. MATLAB is not case sensitive with respect to "built-in" MATLAB functions. For example, the length command can be upper or lower case

```
» LENGTH(x)
ans = 6
```

Notice that we have not named an upper case X variable. See what happens when we try to find the length of `X`

```
» LENGTH(X)
??? Undefined function or variable.
Symbol in question ==> X
```

Sometimes it is desirable to clear all of the variables in a workspace. This is done by simply typing

```
» clear
```

more frequently you may wish to clear a particular variable, such as x

```
» clear x
```

You may wish to quit MATLAB but save your variables so you don't have to retype or recalculate them during your next MATLAB session. To save all of your variables, use

```
» save file_name
```

(Saving your variables does not remove them from your workspace; only **`clear`** can do that)

You can also save just a few of your variables

```
» save file_name x y z
```

To load a set of previously saved variables

```
» load file_name
```

b) Complex variables

Both i and j represent the imaginary number, √-1, by default

```
» i
ans = 0 + 1.0000i
» j
ans = 0 + 1.0000i

» sqrt(-3)
ans = 0 + 1.7321i
```

Note that these variables (`i` and `j`) can be redefined (as the index in a `for` loop, for example), not included in your course.

Matrices can be created where some of the elements are complex and the others are real

```
» a = [sqrt(4), 1;sqrt(-4), -5]

a =  2.0000         1.0000
     0 + 2.0000i  -5.0000
```

Recall that the semicolon designates the end of a row.

c) Some Matrix Operations

Matrix multiplication is straight-forward

```
» b = [1 2 3;4 5 6]
b = 1 2 3
    4 5 6
```

using the `a` matrix that was generated above:

```
» c = a*b
c =
  6.0000              9.0000             12.0000
-20.0000 + 2.0000i -25.0000 + 4.0000i -30.0000 + 6.0000i
```

Notice again that MATLAB automatically deals with complex numbers.

Sometimes it is desirable to perform an element by element multiplication. For example, d(i,j) = b(i,j)*c(i,j) is performed by using the .* command

```
» d = c.*b
d =
 1.0e+02 *0.0600   0.1800            0.3600
-0.8000 + 0.0800i -1.2500 + 0.2000i -1.8000 + 0.3600i
```

Similarly, element by element division, b(i,j)/c(i,j), can be performed using `./`

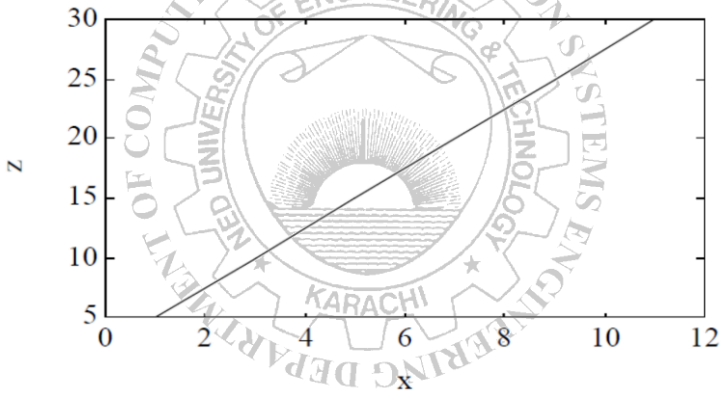
Other matrix operations include: (i) taking matrix to a power, and (ii) the matrix exponential. These are operations on a square matrix

```
» f = a^2
f = 4.0000 + 2.0000i -3.0000
         0 - 6.0000i 25.0000 + 2.0000i
» g = expm(a)
g = 7.2232 + 1.8019i    1.0380 + 0.2151i
   -0.4302 + 2.0760i   -0.0429 + 0.2962i
```

d) Plotting

For a standard solid line plot, simply type

```
» plot(x,z)
```



Axis labels are added by using the following commands

```
» xlabel('x')
» ylabel('z')
```

For more plotting options, type

```
» help plot
```

PLOT Plot vectors or matrices.

`PLOT(X,Y)` plots vector X versus vector Y. If X or Y is a matrix, then the vector is plotted versus the rows or columns of the matrix, whichever lines up. PLOT(X1,Y1,X2,Y2) is another way of producing multiple lines on the plot. PLOT(X1,Y1,':',X2,Y2,'+') uses a dotted line for the first curve and the point symbol + for the second curve. Other line and point types are:

solid -	plus +	red r
dashed --	star *	green g
dotted :	circle o	blue b
dashdot -.	x-mark x	white w
point .	etc...	invisible i

```
PLOT(Y) plots the columns of Y versus their index. PLOT(Y) is
equivalent to PLOT(real(Y),imag(Y)) if Y is complex. In all other
uses of PLOT, the imaginary part is ignored.
See SEMI, LOGLOG, POLAR, GRID, SHG, CLC, CLG, TITLE, XLABEL, YLABEL,
AXIS, HOLD, MESH, CONTOUR, SUBPLOT.
```

Text can be added directly to a figure using the `gtext` command.
`gtext('string')` displays the graph window, puts up a cross-hair, and waits for a mouse button or keyboard key to be pressed. The cross-hair can be positioned with the mouse. Pressing a mouse button or any key writes the text string onto the graph at the selected location.

Consider now the following equation

$$y(t) = 4\, e^{-0.1\, t}$$

We can solve this for a vector of t values by two simple commands

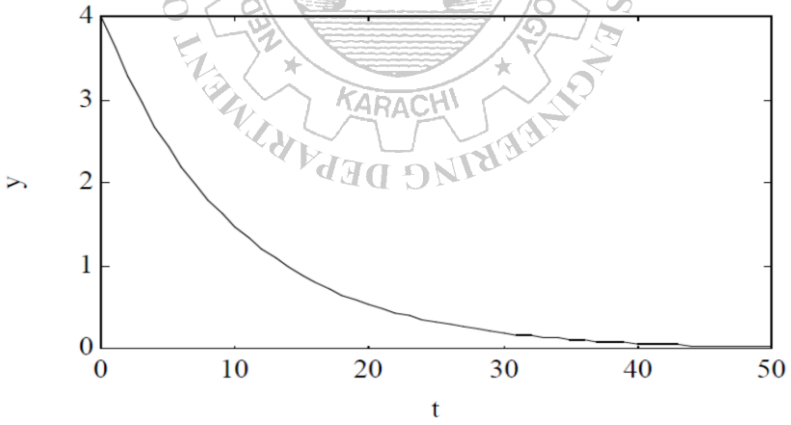
```
» t = 0:1:50;
» y = 4*exp(-0.1*t);
```

and we can obtain a plot by typing

```
» plot(t,y)
```

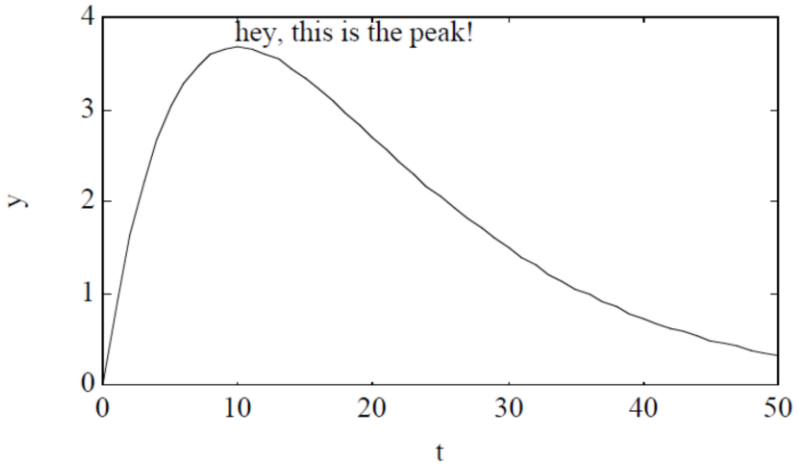
Notice that we could shorten the sequence of commands by typing

```
» plot(t,4*exp(-0.1*t))
```



We can plot the function $y(t) = t\, e^{-0.1\, t}$ by using

```
» y = t.*exp(-0.1*t);
» plot(t,y)
» gtext('hey, this is the peak!')
» xlabel('t')
» ylabel('y')
```



Further functions related to PLOT will be discussed in proceeding lab sessions

e) More Matrix Stuff

A matrix can be constructed from 2 or more vectors. If we wish to create a matrix v which consists of 2 columns, the first column containing the vector x (in column form) and the second column containing the vector z (in column form) we can use the following

```
» v = [x',z']
v =
      1   5
      3   10
      5   15
      7   20
      9   25
      11  30
```

If we wished to look at the first column of v, we could use

```
» v(:,1)
ans =
            1
            3
            5
            7
            9
            11
```

If we wished to look at the second column of v, we could use

```
» v(:,2)
ans =
            5
            10
            15
            20
            25
            30
```

And we can construct the same plot as before, by using ('--' gives a dotted line)

f) m-files

You can create your own matrices using M-files, which are text files containing MATLAB code. Use the MATLAB Editor or another text editor to create a file containing the same statements you would type at the MATLAB command line. Save the file under a name that ends in "`.m`".
For example, create a file containing these five lines.

$$A = \begin{pmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{pmatrix}$$

Store the file under the name `first.m`. Then the statement `first` reads the file and creates a variable, `A`, containing our example matrix.

g) Diary

When preparing homework solutions it is often necessary to save the sequence of commands and output results in a file to be turned in with the homework. The diary command allows this.
`diary file_name` causes a copy of all subsequent terminal input and most of the resulting output to be written on the file named `file_name`. `diary off` suspends it. `diary on` turns it back on. `diary`, by itself, toggles the diary state. Diary files may be edited later with a text editor to add comments or remove mistaken entries.
Often the consultants wish to see a diary file of your session to assist them in troubleshooting your MATLAB problems.

h) Deleting Rows and Columns

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

```
X = A;
```

Then, to delete the second column of `X`, use

```
X(:,2) = []
```

This changes `X` to

$$X = \begin{pmatrix} 16 & 2 & 13 \\ 5 & 11 & 8 \\ 9 & 7 & 12 \\ 4 & 14 & 1 \end{pmatrix}$$

If you delete a single element from a matrix, the result is not a matrix anymore. So, expressions like

```
X(1,2) = []
```

will result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector. So

```
X(2:2:10) = []
```

results in

```
X =16     9     2     7    13    12     1
```

i) Scalar Expansion

Matrices and scalars can be combined in several different ways. For example, a scalar is subtracted from a matrix by subtracting it from each element. The average value of the elements in matrix `A` is 8.5, so

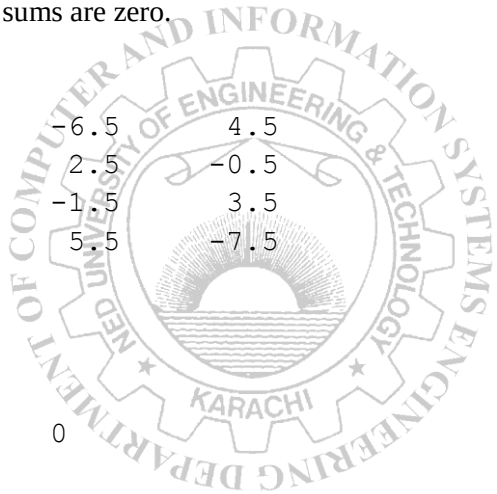
```
B = A - 8.5
```

forms a matrix whose column sums are zero.

```
B =
      7.5     -5.5     -6.5      4.5
     -3.5      1.5      2.5     -0.5
      0.5     -2.5     -1.5      3.5
     -4.5      6.5      5.5     -7.5

sum(B)

ans =
     0     0     0     0
```



With scalar expansion, MATLAB assigns a specified scalar to all indices in a range. For example,

```
B(1:2,2:3) = 0
```

zeroes out a portion of `B`.

```
B =
      7.5        0        0      4.5
     -3.5        0        0     -0.5
      0.5     -2.5     -1.5      3.5
     -4.5      6.5      5.5     -7.5
```

j) Suppressing Output

If you simply type a statement and press Return or Enter, MATLAB automatically displays the results on screen. However, if you end the line with a semicolon, MATLAB performs the computation but does not display any output. This is particularly useful when you generate large matrices. For example,

```
A = ones(100);
```

k) Entering Long Statements

If a statement does not fit on one line, use an ellipsis (three periods) "`...`" followed by `Return` or `Enter` to indicate that the statement continues on the next line.
For example,

```
s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7...
      - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

Blank spaces around the =, +, and − signs are optional, but they improve readability.

Summary of Commonly Used Commands

`clear`	removes all variables from workspace
`clc`	clears command window
`diary`	save the text of a MATLAB session
`exp`	exponential function
`format`	output display format
`function`	user generated function
`gtext`	place text on a plot
`help`	
`hold`	holds current plot and allows new plot to be placed on current plot
`length`	length of a vector
`lookfor`	keyword search on help variables
`plot`	plots vectors
`size`	size of the array
`subplot`	multiple plots in a figure window
`who`	view variables in workspace
`whos`	View variables in workspace, with more detail (size, etc.)
`*`	matrix multiplication
`'`	Transpose
`;`	suppress printing (also - end of row, when used in matrices)
`.*`	element by element multiplication
`./`	element by element division
`:`	denotes a column in a matrix or creates a vector

APPENDIX B

An introduction to SIMULINK

Starting Simulink
Basic Elements
Building a System
Running Simulations

Simulink is a graphical extension to MATLAB for the modeling and simulation of systems. In Simulink, systems are drawn on screen as block diagrams. Many elements of block diagrams are available (such as transfer functions, summing junctions, etc.), as well as virtual input devices (such as function generators) and output devices (such as oscilloscopes). Simulink is integrated with MATLAB and data can be easily transferred between the programs. In this tutorial, we will introduce the basics of using Simulink to model and simulate a system.

Simulink is supported on UNIX, Macintosh, and Windows environments, and it is included in the student version of MATLAB for personal computers. For more information on Simulink, contact the Math Works.

The idea behind these tutorials is that you can view them in one window while running Simulink in another window. Do not confuse the windows, icons, and menus in the tutorials for your actual Simulink windows. Most images in these tutorials are not live - they simply display what you should see in your own Simulink windows. All Simulink operations should be done in your Simulink windows.

Starting Simulink

Simulink is started from the MATLAB command prompt by entering the following command:

```
>>simulink
```

Alternatively, you can click on the "Simulink Library Browser" button at the top of the MATLAB command window.

The Simulink Library Browser window should now appear on the screen. Most of the blocks needed for modeling basic systems can be found in the subfolders of the main "Simulink" folder (opened by clicking on the "+" in front of "Simulink").

1. Basic Elements

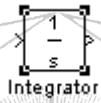
There are two major classes of elements in Simulink: **blocks** and **lines**. Blocks are used to generate, modify, combine, output, and display signals. Lines are used to transfer signals from one block to another.

a. Blocks

The subfolders underneath the "Simulink" folder indicate the general classes of blocks available for us to use:

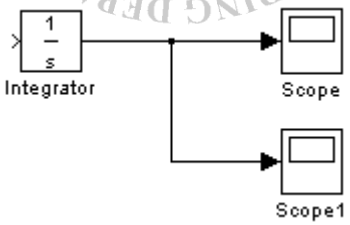
- Continuous: Linear, continuous-time system elements (integrators, transfer functions, state-space models, etc.)
- Discrete: Linear, discrete-time system elements (integrators, transfer functions, state-space models, etc.)
- Functions & Tables: User-defined functions and tables for interpolating function values
- Math: Mathematical operators (sum, gain, dot product, etc.)
- Nonlinear: Nonlinear operators (coulomb/viscous friction, switches, relays, etc.)
- Signals & Systems: Blocks for controlling/monitoring signal(s) and for creating subsystems
- Sinks: Used to output or display signals (displays, scopes, graphs, etc.)
- Sources: Used to generate various signals (step, ramp, sinusoidal, etc.)

Blocks have zero to several input terminals and zero to several output terminals. Unused input terminals are indicated by a small open triangle. Unused output terminals are indicated by a small triangular point. The block shown below has an unused input terminal on the left and an unused output terminal on the right.



b. Lines

Lines transmit signals in the direction indicated by the arrow. Lines must always transmit signals from the output terminal of one block to the input terminal of another block. One exception to this is that a line can tap off of another line. This sends the original signal to each of two (or more) destination blocks, as shown below:

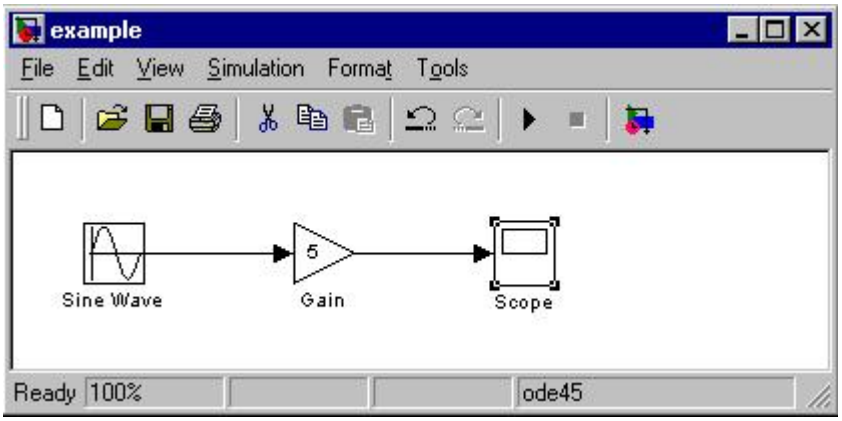


Lines can never inject a signal *into* another line; lines must be combined through the use of a block such as a summing junction.

A signal can be either a scalar signal or a vector signal. For Single-Input, Single-Output systems, scalar signals are generally used. For Multi-Input, Multi-Output systems, vector signals are often used, consisting of two or more scalar signals. The lines used to transmit scalar and vector signals are identical. The type of signal carried by a line is determined by the blocks on either end of the line.

I. Building a System

To demonstrate how a system is represented using Simulink, we will build the block diagram for a simple model consisting of a sinusoidal input multiplied by a constant gain, which is shown below:



This model will consist of three blocks: Sine Wave, Gain, and Scope. The Sine Wave is a **Source Block** from which a sinusoidal input signal originates. This signal is transferred through a **line** in the direction indicated by the arrow to the Gain **Math Block**. The Gain block modifies its input signal (multiplies it by a constant value) and outputs a new signal through a **line** to the Scope block. The Scope is a **Sink Block** used to display a signal (much like an oscilloscope).

We begin building our system by bringing up a new model window in which to create the block diagram. This is done by clicking on the "New Model" button in the toolbar of the Simulink Library Browser (looks like a blank page).

Building the system model is then accomplished through a series of steps:

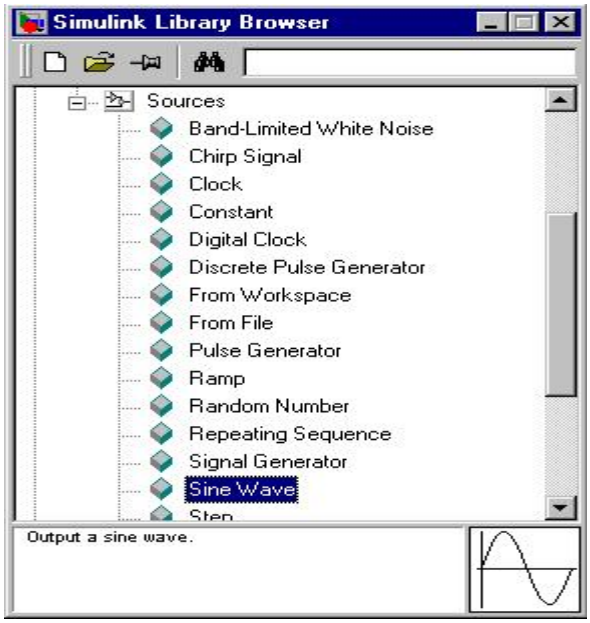
1. The necessary blocks are gathered from the Library Browser and placed in the model window.
2. The parameters of the blocks are then modified to correspond with the system we are modelling.
3. Finally, the blocks are connected with lines to complete the model.

Each of these steps will be explained in detail using our example system. Once a system is built, simulations are run to analyze its behavior.

II. Gathering Blocks

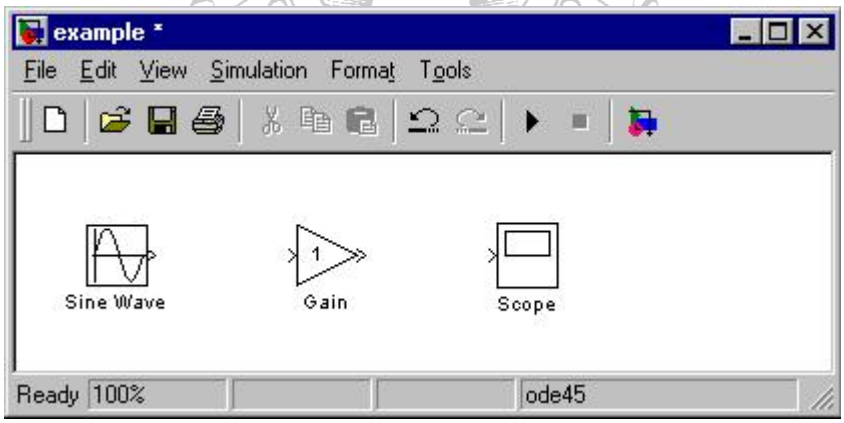
Each of the blocks we will use in our example model will be taken from the Simulink Library Browser. To place the Sine Wave block into the model window, follow these steps:

1. Click on the "+" in front of "Sources" (this is a subfolder beneath the "Simulink" folder) to display the various source blocks available for us to use.
2. Scroll down until you see the "Sine Wave" block. Clicking on this will display a short explanation of what that block does in the space below the folder list:



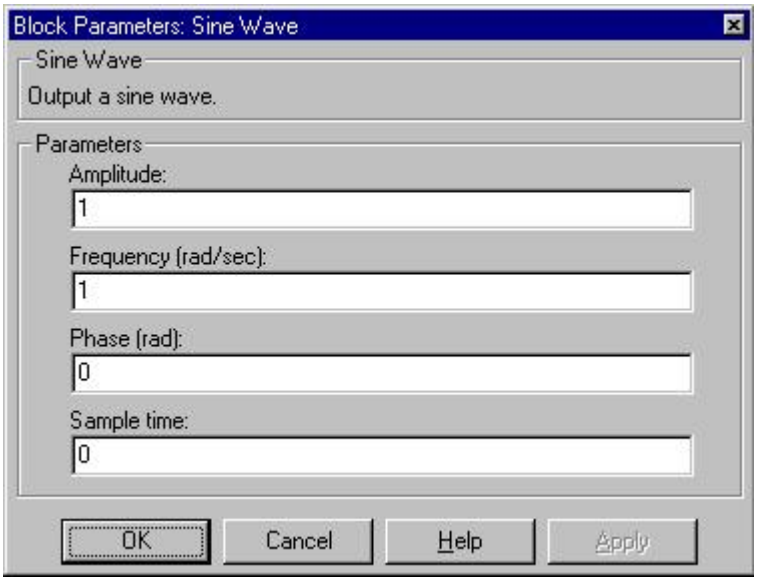
3. To insert a Sine Wave block into your model window, click on it in the Library Browser and drag the block into your workspace.

The same method can be used to place the Gain and Scope blocks in the model window. The "Gain" block can be found in the "Math" subfolder and the "Scope" block is located in the "Sink" subfolder. Arrange the three blocks in the workspace (done by selecting and dragging an individual block to a new location) so that they look similar to the following:



III. Modifying the Blocks

Simulink allows us to modify the blocks in our model so that they accurately reflect the characteristics of the system we are analyzing. For example, we can modify the Sine Wave block by double-clicking on it. Doing so will cause the following window to appear:



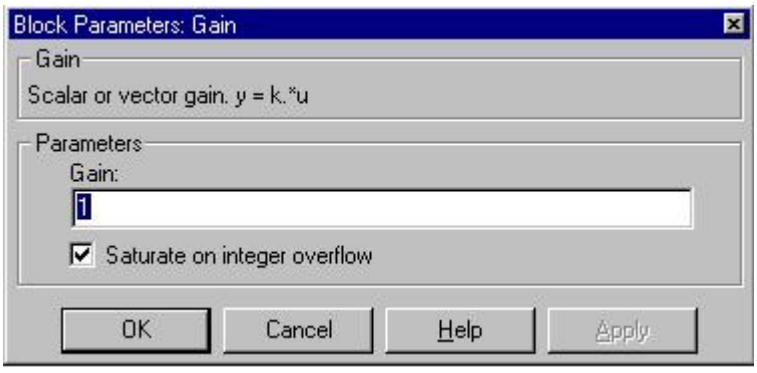
This window allows us to adjust the amplitude, frequency, and phase shift of the sinusoidal input. The "Sample time" value indicates the time interval between successive readings of the signal. Setting this value to 0 indicates the signal is sampled continuously.

Let us assume that our system's sinusoidal input has:

- Amplitude = 2
- Frequency = pi
- Phase = pi/2

Enter these values into the appropriate fields (leave the "Sample time" set to 0) and click "OK" to accept them and exit the window. Note that the frequency and phase for our system contain 'pi' (3.1415...). These values can be entered into Simulink just as they have been shown.

Next, we modify the Gain block by double-clicking on it in the model window. The following window will then appear:



Note that Simulink gives a brief explanation of the block's function in the top portion of this window. In the case of the Gain block, the signal input to the block (u) is multiplied by a constant (k) to create the block's output signal (y). Changing the "Gain" parameter in this window changes the value of k.

For our system, we will let k = 5. Enter this value in the "Gain" field, and click "OK" to close the window.

The Scope block simply plots its input signal as a function of time, and thus there are no system parameters that we can change for it. We will look at the Scope block in more detail after we have run our simulation.

IV. Connecting the Blocks

For a block diagram to accurately reflect the system we are modeling, the Simulink blocks must be properly connected. In our example system, the signal output by the Sine Wave block is transmitted to the Gain block. The Gain block amplifies this signal and outputs its new value to the Scope block, which graphs the signal as a function of time. Thus, we need to draw lines from the output of the Sine Wave block to the input of the Gain block, and from the output of the Gain block to the input of the Scope block.

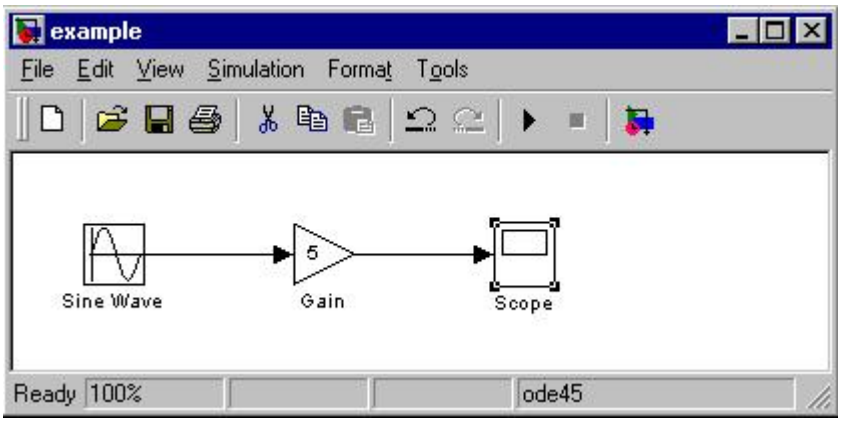
Lines are drawn by dragging the mouse from where a signal starts (output terminal of a block) to where it ends (input terminal of another block). When drawing lines, it is important to make sure that the signal reaches each of its intended terminals. Simulink will turn the mouse pointer into a crosshair when it is close enough to an output terminal to begin drawing a line, and the pointer will change into a double crosshair when it is close enough to snap to an input terminal. A signal is properly connected if its arrowhead is filled in. If the arrowhead is open, it means the signal is not connected to both blocks. To fix an open signal, you can treat the open arrowhead as an output terminal and continue drawing the line to an input terminal in the same manner as explained before.



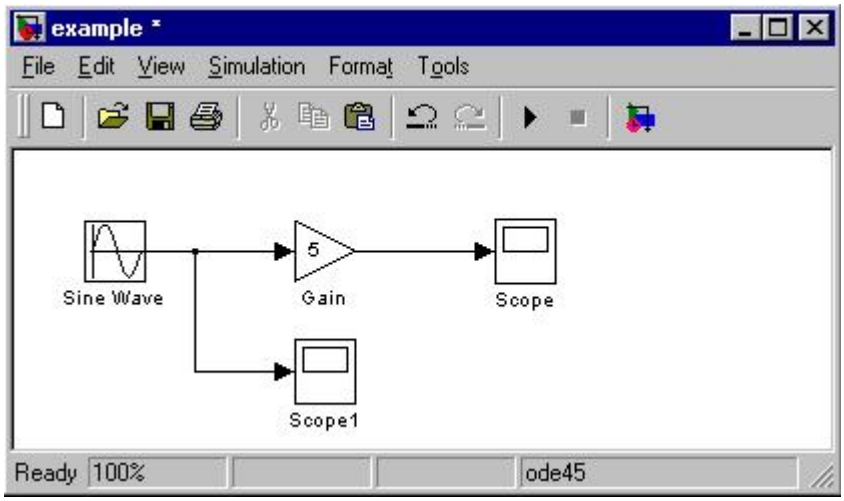
Properly Connected Signal **Open Signal**

When drawing lines, you do not need to worry about the path you follow. The lines will route themselves automatically. Once blocks are connected, they can be repositioned for a neater appearance. This is done by clicking on and dragging each block to its desired location (signals will stay properly connected and will re-route themselves).

After drawing in the lines and repositioning the blocks, the example system model should look like:



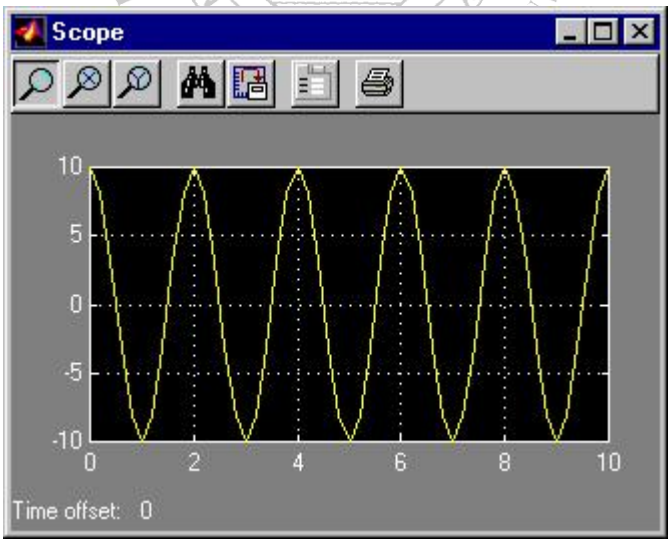
In some models, it will be necessary to branch a signal so that it is transmitted to two or more different input terminals. This is done by first placing the mouse cursor at the location where the signal is to branch. Then, using either the CTRL key in conjunction with the left mouse button or just the right mouse button, drag the new line to its intended destination. This method was used to construct the branch in the Sine Wave output signal shown below:



The routing of lines and the location of branches can be changed by dragging them to their desired new position. To delete an incorrectly drawn line, simply click on it to select it, and hit the DELETE key.

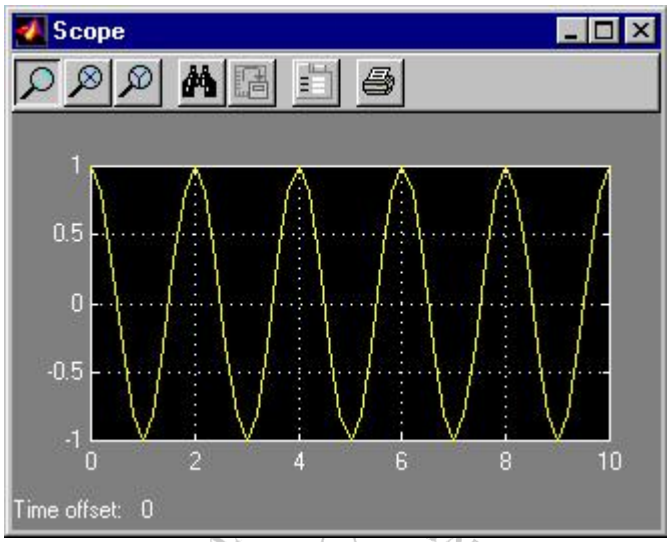
V. Running Simulations

Now that our model has been constructed, we are ready to simulate the system. To do this, go to the **Simulation** menu and click on **Start**, or just click on the "Start/Pause Simulation" button in the model window toolbar (looks like the "Play" button on a VCR). Because our example is a relatively simple model, its simulation runs almost instantaneously. With more complicated systems, however, you will be able to see the progress of the simulation by observing its running time in the lower box of the model window. Double-click the Scope block to view the output of the Gain block for the simulation as a function of time. Once the Scope window appears, click the "Autoscale" button in its toolbar (looks like a pair of binoculars) to scale the graph to better fit the window. Having done this, you should see the following:



Note that the output of our system appears as a cosine curve with a period of 2 seconds and amplitude equal to 10. Does this result agree with the system parameters we set? Its amplitude makes sense when we consider that the amplitude of the input signal was 2 and the constant gain of the system was 5 (2 x 5 = 10). The output's period should be the same as that of the input signal, and this value is a function of the frequency we entered for the Sine Wave block (which was set equal to pi). Finally, the output's shape as a cosine curve is due to the phase value of pi/2 we set for the input (sine and cosine graphs differ by a phase shift of pi/2).

What if we were to modify the gain of the system to be 0.5? How would this affect the output of the Gain block as observed by the Scope? Make this change by double-clicking on the Gain block and changing the gain value to 0.5. Then, re-run the simulation and view the Scope (the Scope graph will not change unless the simulation is re-run, even though the gain value has been modified). The Scope graph should now look like the following:



Note that the only difference between this output and the one from our original system is the amplitude of the cosine curve. In the second case, the amplitude is equal to 1, or 1/10th of 10, which is a result of the gain value being 1/10th as large as it originally was.

TABLE OF SIMULINK BLOCKS

SOURCES	
Clock	Provide and display system time.
Constant	Inject a constant value.
From File	Read data from a file.
From Workspace	Read data from a matrix.
Signal Generator	Generate various waveforms.
Sine Wave	Generate a sine wave.
Step Fcn	Generate a step function.
White Noise	Generate random noise.

SINKS	
Scope	Display signals during a simulation.
To File	Write data to a file.
To Workspace	Write data to a matrix.

DISCRETE	
Discrete State-Space	Discrete state-space system.
Discrete Transfer Fcn	Discrete transfer function.
Discrete Zero-Pole	Discrete system as zeros, poles and gain.
Filter	Implement IIR and FIR filters.
Unit Delay	Delay signal for one sample period.

LINEAR	
Derivative	Output the time derivative of the input.
Gain	Multiply input by a constant.
Integrator	Integrate a signal.
State-Space	Implement a linear state-space system.
Sum	Sum inputs together.
Transfer Fcn	Implement a linear transfer function.
Zero-Pole	Specify a linear system as zeros, poles and gain

NONLINEAR	
Abs	Output the absolute value of the input.
Backslash	Model hysteresis.
Dead Zone	Provide a region of zero output.
Fcn	General purpose function of the input.
Look Up Table	Perform piecewise linear mapping.
MATLAB Fcn	Apply MATLAB function to the input.
Product	Multiply inputs together.
Rate limiter	Limit the rate of change of a signal.
Relay	Switch output between two values.
Saturation	Limit excursion of the signal.
S-function	Make an S-function into a block
Switch	Switch between two inputs.
Transport Delay	Delay the input by a given amount of time.

CONNECTIONS	
Demux	Separate vector signal into scalar signals.
Inport	Provide a link to an external input.
Mux	Group scalar lines into a vector line.
outport	Provide a link to an external output.

EXTRAS	
Sources	Additional sources.
Display Devices	Additional display devices.
Filters	Additional Filters.
Controllers	PID, anti-windup, and optimal controllers.
Linear	Additional linear blocks.
Nonlinear	Additional nonlinear blocks.
Discrete	Additional discrete blocks
Logic	Logic blocks (AND, OR, NAND, etc.)
System ID	System identification blocks.
Robust Control Demos	Robust control demos.
MuTools Demos	Mu analysis of control systems.
Block Library	Most commonly used blocks.
Demos	Simulink demos.